To my loving parents.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Jean-Pierre Talpin. Thanks for all his support, my three years of Ph.D. have been a very useful time. I learnt much from his experience and knowledge. He has been a great advisor. He gave me the freedom of pursing my goals and interests while always providing guidance. Thank you so much Jean-Pierre.

I would like to thank the members of my Ph.D. committee, specially Jean-Paul Bodeveix and Laure Gonnord who agreed on reviewing my dissertation and gave useful and interesting comments.

I also would like to thank my colleagues Loïc Besnard, Thierry Gautier, Paul Le Guernic, Huafeng Yu, Yue Ma, Christophe Junke, Adnan Bouakaz, Sun Ke in the TEA group at INRIA, Abdoulaye Gamatié and Sandeep Shukla for their friendship and support. My special thanks to Thierry and Abdoulaye who has read and commented on every bit of my reports and this dissertation, and who has listened to every idea I had during my time at INRIA.

I also would like to thanks my friends here, in RENNES for their help and relaxed time.

Last but not least, I would like to thank my family, especially my parents, Van Tac & Ngo Ngo who have always been there for me. Only I know that this dissertation cannot be carried out without them.

ABSTRACT

Synchronous languages such as SIGNAL, LUSTRE and ESTEREL are dedicated to designing safety-critical systems. Their compilers are large and complicated programs that may be incorrect in some contexts, which might produce silently bad compiled code when compiling source programs. The bad compiled code can invalidate the safety properties that are guaranteed on the source programs by applying formal methods. Adopting the translation validation approach, this thesis aims at formally proving the correctness of the highly optimizing and industrial SIGNAL compiler. The correctness proof represents both source program and compiled code in a common semantic framework, then formalizes a relation between the source program and its compiled code to express that the semantics of the source program are preserved in the compiled code.

Les langages synchrones tels que SIGNAL, LUSTRE et ESTEREL sont dédiés à la conception de systèmes critiques. Leurs compilateurs, qui sont de très gros programmes complexes, peuvent a priori se révéler incorrects dans certains situations, ce qui donnerait lieu alors à des résultats de compilation erronés non détectés. Ces codes fautifs peuvent invalider des propriétés de sûreté qui ont été prouvées en appliquant des méthodes formelles sur les programmes sources. En adoptant une approche de validation de la traduction, cette thèse vise à prouver formellement la correction d'un compilateur optimisé et industriel de SIGNAL. La preuve de correction représente dans un cadre sémantique commun le programme source et le code compilé, et formalise une relation entre eux pour exprimer la préservation des sémantiques du programme source dans le code compilé.

CONTENTS

LISTINGS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Synchronous programming languages such as SIGNAL, LUSTRE and ESTEREL propose a formal semantic framework to give high-level specification of safety-critical software in automotive and avionics systems [17, 71, 73, 81]. As other programming languages, synchronous languages are associated with a compiler. The compiler takes a source program, analyses and transforms it, performs optimizations, and finally generates executable code in a general-purpose programming language (e.g., C, C^{++}, or JAVA). This compilation process is depicted as in Figure 1.1.

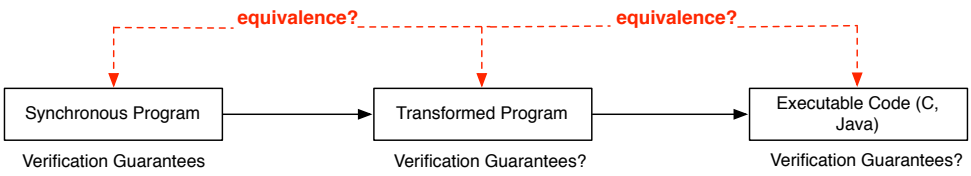

Fig. 1.1 The compilation process of a synchronous compiler

Safety-critical systems are those systems whose failure could result in loss of life, or damage to the environment. Safety-critical systems need to be validated to ensure that their specified safety properties are implemented correctly. Software validation is traditionally done by using testing techniques which, in the case of safety-critical systems, is not sufficient [102]. The validation of safety-critical systems needs to be complemented by the use of formal verification, such as static analysis, model checking, or proof, to guarantee full coverage of the safety requirements on the validated system.

Since synchronous languages are based on strong semantic models, they provide much higher level of abstraction, expressivity, and clarity at source level rather than once compiled into C code. That makes the application of formal methods much simpler to enforce safety properties.

However, a compiler is a large and very complex program which often consists of hundreds of thousands, if not millions, lines of code, divided into multiple sub-systems and modules. Moreover, compiler modules often interact in very complex ways, and the design and implementation of a compiler is a substantial engineering task. The compilation process involves many analyzes, program transformations and optimizations. Some transformations and optimizations may introduce additional information, or constrain the compiled program. They may refine its meaning and specialize its behavior to meet a specific safety or optimization goal.

Consequently, it is not uncommon that compilers silently issue an incorrect result in some unexpected context or inappropriate optimization goal. For example, the GNU Compiler Collection (GCC) is a compiler produced by GNU Project supporting several languages. GCC has been widely used in software development. It is also available for most embedded platforms. Although it has a long history of development and review by many people, no programmer can assert that it is always right in any context; see the list of bugs which have been found at GCC Bugzilla [121]. Considering bug number 15549, for instance, GCC will compile the expression $b < c$ as $(b == 0)\&(c! = 0)$ into the following program, which always aborts.

Listing 1.1 Bug 15549

```
1 int lt(_Bool b, unsigned char c) {
2    return b < c;
3 }
4
5 int main() {
6    if (!lt(1,'a')
7       abort();
8 }
```

An incorrectly compiled program like the one above will most likely invalidate the safety properties that were secured on the source program. Yet, it is natural to require a compiler to guarantee the preservation of the source program's semantics, hence the safety properties are secured on it.

1.1 Correctness of compilation

Proving the correctness of a compiler can be based on the examination of the developed compiler's source code itself, meaning that a qualification process applies on the develop-

ment of the compiler, the source of the compiler, and/or the compiler's output. Qualifying a compiler is rare because of the tremendous administrative effort involved. Qualification amounts to demonstrate compliance with all recommendations and objectives specified in the certification standards for safety-critical softwares: DO-178C and its European equivalent ED-12 [49]. Although DO-178 has been successful in industry, the cost of complying with it is significant: the activities on verification it incurs may well cost seven times more than the development effort needed [126]. A more traditional method is therefore to solely inspect or formally verify the compiler's output. This task requires less unitary effort, but has to be repeated every time a new target code is generated. For instance, the work of Blanchet et al. [23, 24] provides a method to design and implement a special-purpose abstract interpretation based static program analyzer for the verification of safety critical embedded real-time software. And the static program analyzer ASTRÉE [9] aims at proving the absence of *run time errors (RTE)* in the generated C code of the synchronous data-flow compiler from LUSTRE programs. One last resort is hence to formally verify the correctness of the compiler itself.

A compiler is a computer program that reads an input program in one source language and translates it into a semantically equivalent refined program in another target language, Figure 1.2. The structure of a compiler consists of two parts:

Analysis The analysis part creates an intermediate representation of the source program and stores information about the source program in a data structure, the *symbol table*.

Synthesis The synthesis part generates the desired output program from the intermediate representation and information stored in the symbol table.

We usually call the analysis part the *front end* of the compiler, and the synthesis part its *back end*.

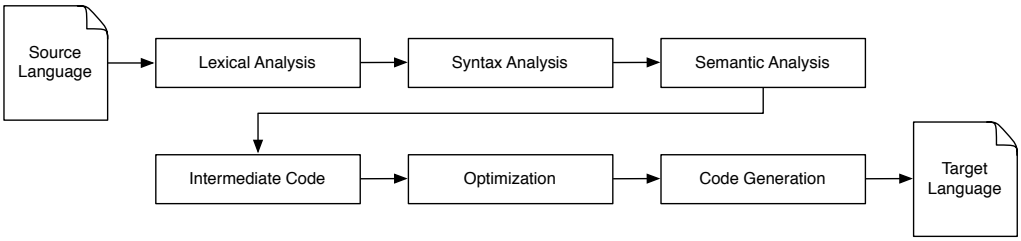

Fig. 1.2 Phases of compiler design

Suppose a specification of the properties that a computer program must satisfy when

executed, e.g., its pre and post-conditions. It may describe interactions and relationships among program components (e.g., program variables, functions,...). The program is said to be correct with respect to its inputs and outputs if for all program executions with inputs satisfying the required specification. This definition applies for a compiler as well. There are two fundamental principles in design and implementation of a compiler which have to be fulfilled [41]:

The compiler must preserve the meaning of the program being compiled.

The compiler must improve the input program in some discernible way.

For example, consider the compilation of an arithmetic expression, onto a stack-machine, in the form of one address code. Compilation assumes the presence of a stack of operands. Operations pop operands from the stack, evaluate them and push the result back onto the stack. For instance, an integer multiplication takes the top two elements from the stack and pushes their product onto the stack. Stack-machine code of the expression $(x+2)*y$ would be as follows:

Listing 1.2 Stack-machine code of $(x+2)*y$

```
1 push x
2 push 2
3 add
4 push y
5 multiply
```

In order to prove the correctness of this compilation, one can develop an interpreter that feeds test operations and parameters to the compiler and interprets the source and compiled program to check output equality. For the stack-machine code generation, evaluating a source expression with test inputs should yield the same result as the value on top of the stack in the interpreted address code.

In this line of research, Sheridan's paper [128] is a good survey on compiler testing. ACE [2] provides the SUPERTEST compiler test and validation suite which is a large collection of self-testing programs. The *certifying compilation* [104] attests that the generated object code satisfies the properties established on the source program by generating concrete evidences along the compilation into object code.

Systematic compiler verification techniques use formal methods. Formal methods advocate the use of a mathematical framework for the specification, development and verification of software and hardware systems. For the purpose of compiler verification, there are two approaches, in general, to prove the software correctness:

Formal compiler verification Specifying the intended behavior of the compiler in a formal specification language and building a proof that the compiler satisfies behavioral equivalence or refinement.

Translation validation Proving that each run of the compiler preserves the semantics of the source program in the generated code.

We shall discuss these two approaches in Section 1.2 and Section 1.3, and their application to the translation validation to a multi-clocked synchronous data-flow compiler in Section 1.4.

1.2 Formal compiler verification

In the context of software systems, formal verification is a problem of proving the correctness of a system with respect to the desired properties, using formal methods. The verification process consists of constructing an abstract mathematical model of the system behavior and of providing a formal proof on that model. There are several mathematical objects which may be used to construct the abstract model of a system, such as finite state machines, labeled transition systems, Petri nets, automaton, process algebra, and formal semantics of programming languages.

Formal verification can be done though many approaches. One such approach is *deductive* verification. It consists of providing deductive proofs that a system behaves in a certain way that is described in the specification, with the aid of either interactive theorem provers (such as HOL [33, 69], ISABELLE [78], or COQ [42]), or an automated theorem prover. It often requires to have knowledge of the system mechanism and why the system works correctly, and then convey this information to the verification process.

Another approach is *model checking* [37–39, 123]. It involves building an abstract model of the system and ensure that the system model complies with specified requirements by exploring all its accessible states. The system to be verified is often represented in *temporal logics*, such as *Linear Temporal Logic* (LTL) or *Computational Tree Logic* (CTL). The verification process produces a confirmation that the system model conforms to requirements or a counterexample that can be used to locate and eliminate an error. The main disadvantage of this approach is that it does not in general scale to large systems due to the *state explosion* problem. Some techniques must be used to deal with this problem including *abstract interpretation*, *symbolic simulation* and *abstract refinement* [43–45].

A new variant of model checking is inspired by recent advances in efficiently solving

propositional satisfiability problems (or SAT). *Bounded model checking* (BMC) [20, 36] encodes the fact that potential executions of the system model do not conform to the specification in incremental fashion as propositional satisfiability formulas. The bounded number of evaluation steps is increased as long as the resulting propositional formula is satisfiable. Then a concrete counterexample can be extracted as a trace of system states leading to an error state in the system model.

Another approach is to use *inductive* reasoning to prove that a system conforms to its specification. The advances in solvers based on *Satisfiability Modulo Theories* (SMT) have been useful in checking systems inductively. With these solvers, systems can be modeled efficiently, require fewer limitation on representation of the specifications, while still meeting significant performance. In inductive approach, the transition relation of system a property are encoded as logic formulas. Then, it checks that the property is satisfied at initial state as the base case. If the base case holds true, take the assumption that the property holds for some state and prove that it holds for next state as well.

As usual, formal compiler verification is the problem of proving that the behavior of a compiler meets certain specifications. For instance, a compiler of arithmetic expressions for stack machines has to verify that the result of an evaluated operation is the same as that on top of the machine code's stack after execution. A formal compiler verification consists of establishing the given correctness property between source program and its compiled program. In our case, a correctness property should be that, if a source program has well-defined semantics, then it should be observationally equivalent to its generated code. Establishing this correctness property usually consists of:

- Specifying the intended behavior of a compiler in a specification language. This language is defined deterministically based on formal, deductive logic. The specification of the compiler is expressed in terms of the representation of the source and the compiled programs.

- Building a proof, based on some mathematical reasoning and automated proving techniques, to show that the compiler satisfies its model of intended behavior.

Given a source program A, the compilation of a compiler can be considered as a function Cp from a set of source programs to the set of compiled programs and the compilation error: $P_s \longrightarrow P_c \cup \{\texttt{Error}\}$. We denote the compiled program of A by $Cp(A) = C$ and the compilation error by $Cp(A) = \texttt{Error}$. In other words, for every source program, the output of the compilation is either a compiled program or an error. Then the semantic equivalence between the compiled program and the source program is ensured by the correctness of the

compiler. Following the above conception of formal compiler verification, a compiler is a formally verified compiler if it is accompanied with a formal proof of the following theorem [92].

$$\forall A \in P_s, C \in P_c,\ Cp(A) = C \Rightarrow Correct(A,C) \tag{1.1}$$

where $Correct(A,C)$ denotes the correctness property between the source program A and its compiled program C.

Therefore, formal compiler verification formally ensures the correctness of the compiler. It makes sure that all guarantees obtained on the source program are preserved in the compiled program. This is particularly important if the compiler is used in development of safety-critical embedded systems, since the safety requirements for the development tools (e.g., compilers, translators,...) that translate programs are very high and require the most rigorous verification methods.

1.3 Translation validation

In the translation validation approach, the compiler is not verified. Instead, a validator is associated with the compiler to verify the correctness of each run of the compiler.

The notion of translation validation was first introduced by Pnueli et al. in [118] as an approach to verify the correctness of translators (compilers, code generators). The main idea of translation validation is that instead of proving the correctness of the translator, each individual translation (e.g., run of the code generator) is followed by a validation process which checks that the target program correctly implements the source program. It first constructs the formal models of both the source and compiled programs capturing their semantics. Then, it tries to establish a *refinement* relation between the formal models of the source and target program. If the compiled program behaves differently than the source program then one cannot establish that relation. Fortunately, the compilation scenario should provide a counter-example to help correcting the compiler error.

A verification framework which adopts translation validation benefits from the following features:

- The verification framework does not modify or instrument the compiler. It treats the compiler as a “black box” (as long as there is no error in it). It only considers the input program and its compiled result. Hence, it is not affected by updates and modifications to the compiler, as long as its data-structures remain the same.

- In general, the validator is much simpler and smaller than the compiler. Thus, the proof of correctness of the validator takes less effort than the proof of the compiler.
- The verification process is fully automated.
- The validator can be scaled to large programs, in which we represent the desired program semantics with our scalable abstraction and use efficient techniques to achieve the expected goals: traceability and formal evidence.

A verification framework which adopts the translation validation approach consists of the components depicted in Figure 1.3.

Model builder The model builder defines a common semantic representation to capture the semantics of the source program and the compiled program of the translator. The outputs of this module are the formal models of these programs (e.g., they can be a kind of labeled transition system, or a first-order logic formula).

Analyzer The analyzer first formalizes the notion of "correct implementation" as a refinement relation. This relation expresses that the semantics of the source program is preserved during the compilation. This relation is defined based on the common semantic representation of the model builder. The analyzer also provides an automated proof method which allows to prove the existence of the refinement between the formal models. If the analyzer successfully proves the existence of the refinement, a *proof script* will be created. Otherwise, it will generate a *counter-example*.

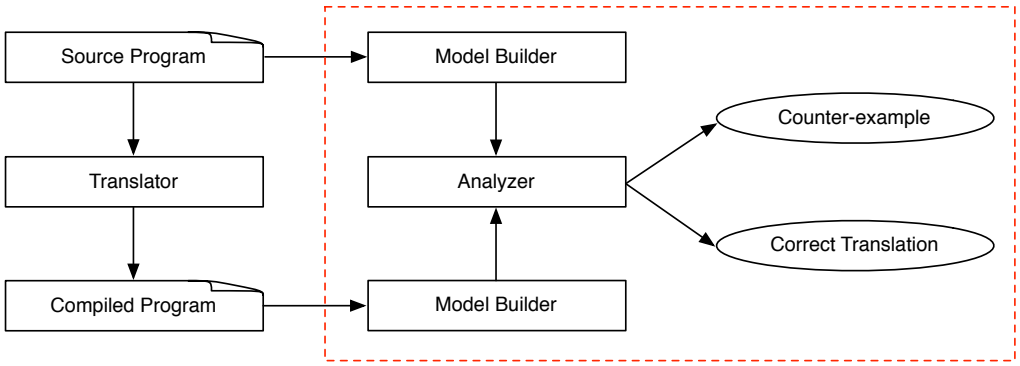

Fig. 1.3 A bird's-eye view of translation validation framework

For example, in the work of Pnueli et al., the semantics of the source program and its compiled program are represented as *Synchronous Transition Systems* (STS). That is the

common semantic framework. Given two STSs, they formalize the concept of "correct translation" as a refinement relation for them which expresses that the semantics of the source program is preserved in the compiled program. The refinement is checked by the use of a solver and the proof script is also generated.

1.4 Contributions

Considering the compiler of the synchronous data-flow language SIGNAL, the compilation process can be divided into three phases as depicted in Figure 1.4. It consists of a sequence of code transformations and optimizations. Some transformations are optimizations that rewrite the code to eliminate inefficient expressions. The transformations may be seen as a sequence of morphisms rewriting SIGNAL programs to SIGNAL programs, meaning that the intermediate representations produced by the compiler are written in SIGNAL. The final steps, C or JAVA code generation, are simple morphisms over the ultimately transformed program.

Clock calculation and Boolean abstraction Calculates the clock of all signals in the program and defines a Boolean abstraction of the program. The clock of a signal defines exactly when a signal shall be evaluated in a program. The intermediate representation of this phase is written in SIGNAL language.

Static scheduling Based on the clock information and the Boolean abstraction obtained at the first stage, the compiler constructs the *Conditional Dependency Graph* (CDG) to represent the static schedule of all signals' evaluation.

Code generation The clocked and scheduled Signal program is ready to directly generate executable code (e.g., sequential code in C or JAVA).

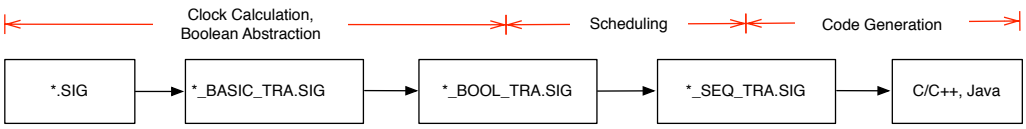

Fig. 1.4 The compilation process of the SIGNAL compiler

As a contribution to this dissertation, we evaluate the concept of translation validation and study the way to adopt it to prove the correctness of the SIGNAL compiler. We also

show that it is possible to construct a validator that is plugged to the compiler in such a way it can provide formal correctness guarantees comparatively strong to these which could be obtained by proof assisted formal compiler verification.

Write C to denote the compiled program and `Error` to denote the compilation error of a source program A. Consider a validator Val which adopts the translation validation approach. The validator can be represented as a function from the set of pairs of a source program and its compiled program to the set of Boolean values: $P_s \times P_c \longrightarrow \mathbb{B}$. The validator that we want to build satisfies the following property:

$$\forall A \in P_s, C \in P_c, Cp(A) = C, Val(A,C) = \texttt{true} \Rightarrow Correct(A,C) \tag{1.2}$$

We now associate each run of the compiler Cp with the validator Val. The following function Cp_{Val} defines a formally verified compilation process from P_s to P_c and the compilation error.

$$Cp_{Val}(A) = \begin{cases} C & \text{if } Cp(A) = C \text{ and } Val(A,C) = \texttt{true} \\ \texttt{Error} & \text{if } Cp(A) = C \text{ and } Val(A,C) = \texttt{false} \\ \texttt{Error} & \text{if } Cp(A) = \texttt{Error} \end{cases}$$

The verification of the derived compiler Cp_{Val} reduces to the verification of the associated validator Val, meaning that the compiler does not need to be verified and can be considered as a black box. The following trivial theorem is the base of our line of work in this dissertation.

Theorem 1 *If the validator Val satisfies the property 1.2, then the derived compiler Cp_{Val} is formally verified in the sense of Theorem 1.1.*

It is obvious to prove globally that the source program and its final compiled program have the same semantics. However, we believe that a better approach is to separate concerns and prove each analysis and transformation stage separately with respect to ad-hoc data-structures to carry the semantic information relevant to that phase.

In the case of the SIGNAL compiler, the preservation of the semantics can be decomposed into the preservation of clock semantics at the *clock calculation* phase and that of data dependencies at the *static scheduling* phase, and, finally, value-equivalence of variables at the *code generation* phase.

Figure 1.5 shows the integration of this verification framework into the compilation process of the SIGNAL compiler. For each phase, the validator takes the source program and its compiled counterpart, and constructs the corresponding formal models of the programs.

Then, it checks the existence of the refinement relation to prove the preservation of the considered semantics. If the result is that the relation does not exist then a "compiler bug" message is emitted. Otherwise, the compiler continues its work.

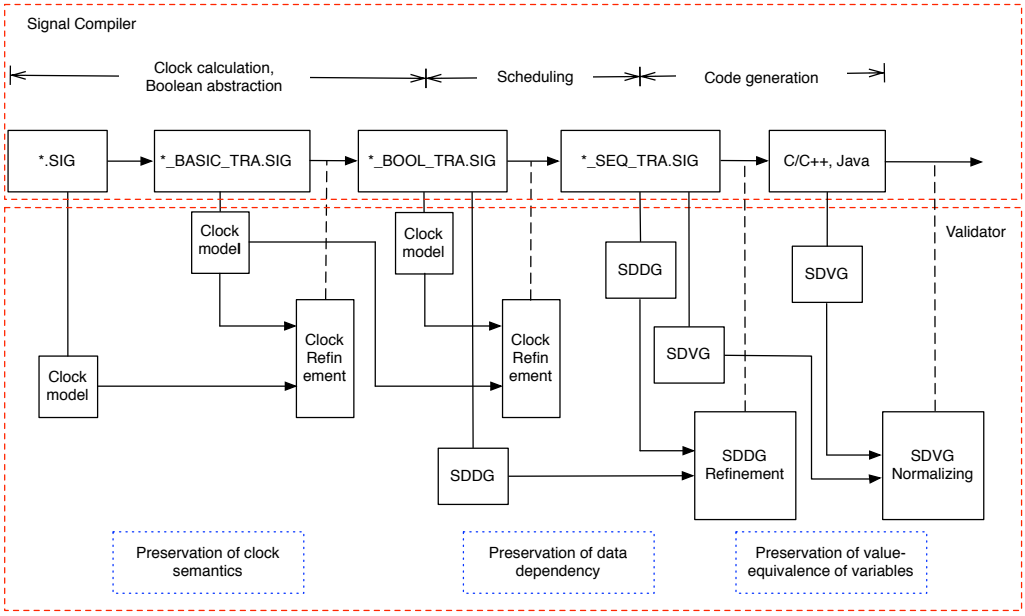

Fig. 1.5 The translation validation for the SIGNAL compiler

1.4.1 Preservation of clock semantics

The first verification stage focuses on proving that all clock relations associated with *signals* in the source and transformed program are equivalent. We propose two approaches to implement this verification, one based on model checking and the other on SMT solving.

In the first approach, the clock semantics of the source and transformed programs are represented by *Polynomial Dynamical Systems* (PDSs). A PDS is a system of equations, in which the coefficients of equations range over $\mathbb{Z}/_3\mathbb{Z}$. The notion of "correct implementation" is formalized as a PDS *refinement*. An automated proof method based on the simulation technique is provided for checking the existence of this refinement.

In the second approach, the clock semantics of the source and transformed programs are formally represented as *clock models*. A clock model is a first-order logic formula that characterizes the presence/absence status of all signals in a SIGNAL program at a given instant. Given two clock models, a *clock refinement* between them is defined which expresses the semantic preservation of clock semantics. A method to check the existence of clock

refinement is defined as a satisfiability problem which can be automatically and efficiently proved by an SMT solver.

Let Cp^{sig} and Val_{clk} be the functions which define the SIGNAL compiler and a validator, respectively. The following function defines a formally verified compiler for the *clock calculation and Boolean abstraction* phase. We write $C \sqsubseteq_{clk} A$ to denote that there exists a refinement between A and C.

$$Cp^{sig}_{Val_{clk}}(A) = \begin{cases} C & \text{if } Cp^{sig}(A) = C \text{ and } Val_{clk}(A,C) = \texttt{true} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = C \text{ and } Val_{clk}(A,C) = \texttt{false} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = \texttt{Error} \end{cases}$$

where $Val_{clk}(A,C) = \texttt{true}$ if and only if $C \sqsubseteq_{clk} A$.

1.4.2 Preservation of data dependency

Given two signals in a source program, the aim of this work is to prove that data dependenies between them are preserved in the scheduled program. This verification also ensures that if there is no *deadlock* in the source program, then there is no deadlocks in the scheduled one either.

In order to do that, the data dependencies among signals are represented by a common semantic framework, called *Synchronous Data-flow Dependency Graph* (SDDG). A SDDG is a labeled directed graph, in which each node is a signal or a clock and each edge represents the dependency between nodes. Each edge is labeled by a clock expression called *clock constraint* at which the dependency between two extremity nodes is effective. The notion of "correct implementation" is formalized as a *dependency refinement* relation between graphs. This relation expresses the semantic preservation of data dependencies. It is implemented using an SMT solver to check the existence of the refinement relation.

Let Val_{dep} be the function which defines a validator. The following function defines a formally verified compiler for the *static scheduling* phase of the SIGNAL compiler. We denote the fact that C refines A by $C \sqsubseteq_{dep} A$.

$$Cp^{sig}_{Val_{dep}}(A) = \begin{cases} C & \text{if } Cp^{sig}(A) = C \text{ and } Val_{dep}(A,C) = \texttt{true} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = C \text{ and } Val_{dep}(A,C) = \texttt{false} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = \texttt{Error} \end{cases}$$

where $Val_{dep}(A,C) = \texttt{true}$ if and only if $C \sqsubseteq_{dep} A$.

1.4.3 Preservation of value-equivalence of variables

This work focuses on proving that every output signal in the source program and the corresponding variable in the compiled program, the generated C program, have the same values. The computations of all signals and their compiled counterparts are represented by a shared value-graph, called *Synchronous Data-flow Value-Graph* (SDVG).

Given a SDVG, assume that we want to show that two variables have the same value. We simply need to check that they are represented by the same sub-graph, meaning that they point to the same graph node. If all output signals in the source program A and the corresponding variables in the generated C program have the same value, then we say that C refines A, denoted by $C \sqsubseteq_{val} A$.

Let Val_{val} be the function which defines a validator. The following function defines a formally verified compiler for the *code generation* phase.

$$Cp^{sig}_{Val_{val}}(A) = \begin{cases} C & \text{if } Cp^{sig}(A) = C \text{ and } Val_{val}(A,C) = \texttt{true} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = C \text{ and } Val_{val}(A,C) = \texttt{false} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = \texttt{Error} \end{cases}$$

where $Val_{val}(A,C) = \texttt{true}$ if and only if $C \sqsubseteq_{val} A$.

1.4.4 Towards a formally verified SIGNAL compiler

The derived SIGNAL compiler that is associated by the validator, denoted by Val, in Figure 1.5 can be defined by the following function from P_s to P_c and the compilation error.

$$Cp^{sig}_{Val}(A) = \begin{cases} C & \text{if } Cp^{sig}(A) = C_{clk}, Cp^{sig}(C_{clk}) = C_{dep}, Cp^{sig}(C_{dep}) = C \text{ and} \\ & Val(A,C) = \texttt{true} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = C_{clk}, Cp^{sig}(C_{clk}) = C_{dep}, Cp^{sig}(C_{dep}) = C \text{ and} \\ & Val(A,C) = \texttt{false} \\ \texttt{Error} & \text{if } Cp^{sig}(A) = \texttt{Error} \text{ or } Cp^{sig}(C_{clk}) = \texttt{Error} \text{ or} \\ & Cp^{sig}(C_{dep}) = \texttt{Error} \end{cases}$$

C_{clk} and C_{dep} are the intermediate forms of the source program A as the outputs of the *clock calculation and Boolean abstraction* and *static scheduling* phases. C is the generated C program from the intermediate form C_{dep}. $Val(A,C) = \texttt{true}$ if and only if $C_{clk} \sqsubseteq_{clk}$

$A, C_{dep} \sqsubseteq_{dep} C_{clk}$ and $C \sqsubseteq_{val} C_{dep}$. The pseudo-code implementation of the function above is given in Listing 1.3.

Listing 1.3 Pseudo-code implementation of formal verified SIGNAL compiler

```
1  if (Cp^sig(A) == Error) return Error;
2  else
3  {
4     if (C_clk ⊑_clk A)
5     {
6        if (Cp^sig(C_clk) == Error) return Error;
7        else
8        {
9           if (C_dep ⊑_dep C_clk)
10          {
11             if (Cp^sig(C_dep) == Error) return Error;
12             else
13             {
14                if (C ⊑_val C_dep) return C;
15                else return Error;
16             }
17          }
18          else return Error;
19       }
20    }
21    else return Error;
22 }
```

1.5 Chapter plan

In the remainder of this dissertation, the chapter structure is as follows:

- Chapter 2 surveys related works in the field of verification of the correctness of the compilation process.
- Chapter 3 introduces the concept of synchronous programming. We study the SIGNAL language as an instance of multi-clocked synchronous data-flow languages used to describe reactive systems, specially safety-critical systems.
- Chapter 4 presents our methods of proving the preservation of clock semantics when the SIGNAL compiler calculates the clock information and makes the Boolean abstraction.

- Chapter 5 provides a method to prove the preservation of data dependencies. We define the common semantic frameworks to capture the data dependencies among variables in the programs, called *Synchronous Data-flow Dependency Graph*. Given the formal representations of data dependencies, we formalize the notion of "correct implementation" as a refinement relation between graphs.
- Chapter 6 presents a method based on the translation validation approach to prove the preservation of value-equivalence of variables between a source SIGNAL program and its generated C code. The computation of variables in the programs is represented as a *Synchronous Data-flow Value-Graph*. Value-equivalence of variables is validated by the *normalization* of graphs.
- Chapter 7 summarizes our work and details some directions of future research.

CHAPTER 2

RELATED WORK IN COMPILER VERIFICATION

The story of compiler verification began in 1967 with Mc.Carthy and Painter [99] presenting the correctness proof for an algorithm compiling arithmetic expressions into machine language, and then with Milner's mechanized logical proof of a compiler in [101].

Mc.Carthy and Painter proposed a method to prove the correctness of a simple compiling algorithm. The input language of the compiler contains arithmetic expressions formed from constants and variables. The expressions allow only one operator, addition. The compiled code is written in an assembly-like language whose instructions are *li* (load immediate), *load*, *sto* (store), and *add*. For example, $(x+3)+(x+(y+2))$ is compiled into the following assembly code:

```
1 load x
2 sto t
3 li 3
4 add t
5 sto t
6 load x
7 sto t + 1
8 load y
9 sto t + 2
10 li 2
11 add t + 2
12 add t + 1
13 add t
```

The semantics of the source language is given by the following formula:

$$\begin{aligned} value(e,\xi) = \text{if } isconst(e) \text{ then } val(e) \\ \text{else if } isvar(e) \text{ then } c(e,\xi) \\ \text{else if } issum(e) \text{ then } value(s1(e),\xi) + value(s2(e),\xi) \end{aligned}$$

where $isconst(e), isvar(e)$ and $issum(e)$ are predicates that check that an expression e is a constant, a variable or the sum of two expressions, and ξ is a *state vector*. The state vector associates the registers of the machine with their content. There are two functions on state vectors: $c(x,\eta)$ denotes the content of register x in machine state η. $a(x,\alpha,\eta)$ denotes the state vector that is obtained from the state vector η by updating the content of register x to α, leaving other registers unchanged. Two state vectors η_1 and η_2 are equal except for variables in A, written $\eta_1 =_A \eta_2$ if $x \notin A, c(x,\eta_1) = c(x,\eta_2)$.

In the same way, the semantics of the object code is given as follows. It describes the state vector η that results from executing an instruction.

$$\begin{aligned} step(s,\eta) = \text{if } isli(s) \text{ then } a(ac, arg(s), \eta) \\ \text{else if } isload(s) \text{ then } a(ac, c(adr(s),\eta), \eta) \\ \text{else if } issto(s) \text{ then } a(adr(s), c(ac,\eta), \eta) \\ \text{else if } isadd(s) \text{ then } a(ac, c(adr(s),\eta) + c(ac,\eta), \eta) \end{aligned}$$

where $isli(s), isload(s), issto(s)$ and $isadd(s)$ are predicates to check that an instructor s is *load immediate*, *load*, *store*, and *add*, respectively. And the state vector that results from executing the program p with the state vector η, is given as follows:

$$\begin{aligned} outcome(p,\eta) = \text{if null(p) then } \eta \\ \text{else } outcome(rest(p), step(first(p),\eta)) \end{aligned}$$

The compiler is defined by a function from the abstract syntax of the source language to the abstract syntax of the object code. The semantics of the compiling algorithm is given by the following formula:

$$\begin{aligned} compile(e,t) = \text{if } isconst(e) \text{ then } mkli(val(e)) \\ \text{else if } isvar(e) \text{ then } mkload(loc(e,map)) \\ \text{else if } issum(e) \text{ then } compile(s1(e),t) * mksto(t) + compile(s2(e),t+1) * mkadd(t) \end{aligned}$$

Symbol t stands for the index of a register in the state vector. All variables are stored in the state vector at indexes less than t, so that registers at index t and above are available for temporary storage. Instructions are named $mkli, mkload, mksto$ and $mkadd$. The operation $p_1 * p_2$ denotes the program obtained by appending sub-program p_2 at the end of p_1.

Assume that there is a map $loc(e, map)$ which associates each expression e to a location in the memory map of the machine. The compiling algorithm is correct if executing the compiled program puts ends up with the value of the compiled expression in the accumulator. No registers except the accumulator and those with addresses $\geq t$ should be affected. This can be expressed as follows:

$$\text{If } c(loc(v,\eta)) = c(v,\xi) \text{ then}$$
$$outcome(compile(e,t),\eta) =_t a(ac, value(e,\xi),\eta).$$

This work illustrates the main steps to formally prove the correctness of a simple compiler. First, one needs to formalize the semantics of the source language and compiled code, and then the compilation algorithm. Then, the definition of correct compilation has to be specified and a theorem proved to demonstrate that it is satisfied by the algorithm.

Since this pioneering work, many proofs of program correctness have been published, using techniques based on testing or on formal methods; see [46] for a detailed survey.

2.1 Compiler verification based on testing

Trust regarding the correctness of a compiler's translation of source code into object code can be established by the validation of the compiler itself or by testing the compiler's output. For instance, the SUPERTEST suite [2] is one of the most comprehensive test and validation suite to verify compilers. It contains a large collection of more than 3 millions test programs. These program are self-testing when compiled and run. They are be divided into the following classes:

- *Conformance test*: to test compilation of the basic constructs of the programming language.
- *Early compiler bugs*: derived from bugs found in earlier compilers that tend to occur repeatedly.
- *Negative tests*: to ensure that compilers handle errors in programs accurately.

- *Stress tests*: to seek the limits of the compiler in terms of sizes, number of basic blocks, live variables, etc., that the compiler handles.
- *Generated tests*: to systematically exploring combinations of operators, types, storage classes and constant values.
- *Tests for the* C^{++} *Standard Library*: These tests verify both conformity with the standard [1, 130] and correctness of the implementation.

The tool interprets the test set definition and test run parameters. Then, it feeds the tests to the compiler. It might run the resulting programs to assert that the compiler passed by the test. As an example, the self-testing program of the SUPERTEST suite in Listing 2.1 checks the correctness of bit clear [89] implementation of a C^{++} compiler.

Listing 2.1 Bit clear test case with SUPERTEST

```
#define COUNT   6
int clear[COUNT] = {2, 3, 5, 7, 11, 13};

void test_bitclear(void) {
   int i, bits, c;
   bits = 0xffff;

   for (i = 0; i < COUNT; i++) {
      c = clear[i];
      bits &= ~(1 << c);
   }

   /* the assertion */
   CVAL_VERIFY(bits == 0xd753);
}
```

The correctness of the compiler's translation of source code into object code can alternatively be based on the examination of the compiler's output through the methodology proposed by the DO-178 standard. This provides user a way to demonstrate that the properties established in the source code still hold in the object code. In order to fulfil this objective, DO-178 recommends a proof that requirements at source-code level can be traceable down to the object code [55, 90], including the integration of software onto its hardware execution platform.

The form of verification which is required by DO-178 is based on high-level requirements, such as "HLR1: the program is never in error state $E1$", and low-level requirements,

such as "LLR1: function F computes output $O1,...,On$ from inputs $I1,...,Im$". For both HLRs and LLRs, the DO-178 guidance requires compliance and robustness verification. These verifications can be done either by testing or by using formal verification as indicates the new standard, DO-178C, also known as DO-333 [50].

The compliance verification focuses on the intended nominal behavior of a compiler. The robustness verification focuses on the behaviors outside the nominal behaviors (e.g., the compilers are free of runtime errors such as out-of-range array elements, deallocation of null pointer, over-flow).

The new version of DO-178 permits to replace part of testing with formal verification. For instance, AIRBUS uses formal analysis tools to compute the worst case execution time and maximum stack usage of executable programs [56] in order to comply with some DO-178 requirements. Compliance and robustness verifications can also be proved by using formal methods. For instance, HLR1 can be expressed as a temporal logic formula on traces of execution and an observer tool can check that the error state is unreachable.

2.2 Compiler verification based on formal methods

There are two approaches, in general, to prove the correctness of a compiler using formal methods. One approach consists of specifying the intended behavior of the compiler in a specification language as a formal model and of building a proof to show that the compiler behaves exactly as prescribed by requirements. The second approach consists of examining the source and compiled programs in order to prove that, for each run of the compiler, the semantics of the source program is preserved. Many correctness proofs of compiler implementations based on the two above approaches have been carried out, formal verification of the compiler itself [27, 35, 120, 133] or the verification of its compiled code [75, 85, 91, 105, 118, 119, 125, 140].

A recent and typical example of compiler correctness proof is [27]. In this example, the correctness of the whole *Iterated Register Coalescing* (IRC) algorithm [67] is formally verified. The verification process works in cooperation with the proof assistant COQ. The register allocation via graph coloring which was invented by Chaitin et al. [34] is widely used in compiler implementation. However, since IRC was published in 1996, several mistakes have been reported in some of its implementations.

The input of IRC is an *interference graph* and a *palette* of colors, the output is the colored graph. In [27], the interference graph is first defined in a purely functional language, GALLINA, and implemented in the COQ prover. Then IRC is written in GALLINA. The

implementation of the abstract interference graph and the operations of the algorithm are formally proved to be correct. The verified program is translated automatically into OCAML code that can be plugged in the COMPCERT compiler to provide correct register allocation.

A compiler is a large and very complex program which often consists of hundreds of thousands, if not millions, lines of code, and is divided into multiple sub-systems and modules. In addition, each compiler implements a particular algorithm in its own way. Consequently, that makes two main drawbacks of the formal verification of the compiler itself approach. First, constructing the specifications of the actual compiler implementation is a long and tedious task. Second, the correctness proof of a compiler implementation, in general, cannot be reused for another compiler.

To deal with this drawbacks of formally verifying the compiler itself, one can prove that the source program and the compiled program are semantically equivalent, which is the approach of *translation validation*. The principle of translation validation is as follows: for a given input sample, the source and the compiled programs will give corresponding *execution traces*. These traces are equivalent if they have the same *observation*. An observation is a sequence (finite or infinite) of values (e.g., values of variables, arguments, returned values,...). The compilation is correct if for any input, the source and the compiled programs have observationally equivalent execution traces.

A pioneering contribution to this area was the work of Pnueli et al. [118, 119] to prove the correctness of the code generator from SIGNAL programs to C programs. Pnueli et al. formalize the semantics of a SIGNAL program and the generated C code in terms of *Synchronous Transition Systems* (STS). A STS consists of the set of states, the set of initial sates and a transition relation. A running of program is represented by a *computation* of STS which is an infinite sequence of states $\sigma = \langle s_0, s_1, s_2, ... \rangle$ such that s_0 is an initial state and s_{i+1} is the successor state of s_i, for all $i \in \mathbb{N}$. And the set of all possible computations represents the semantics of the program. Given a computation σ, an *observation* is an infinite sequence of values by applying the *observation function* on each state of σ. Then, the authors formalize the concept of "correct translation" as a refinement between two STSs which expresses that the semantics of the source program is preserved at the compiled program, meaning that for any observation of the STS of the compiled program, it is also the observation of the STS of the source program. The refinement is generated as a set of verification conditions, and it is proved by the use of a solver such as SMT solver.

Zuck et al. [95, 117, 139, 140] introduce a methodology to validate optimizations by generating a set of verification conditions and using a theorem prover. The main idea of their work is that the validator generates a set of verification conditions based on an invariant

for intra-procedural optimizations. The invariant for an intra-procedural optimization is composed of:

- A relation between the nodes in the control-flow graphs.
- A relation between the program states (e.g., contents of registers, stacks, heaps,...).
- Invariants for the individual input and output programs.

This set of verification conditions indicates the program equivalence for finite slices of program executions. That implies that the optimized program is a correct refinement of the input program.

A representative example is the CompCert project [40]. The CompCert compiler is a formally verified compiler for C language. The compiler is mostly written in the functional programming language Gallina. The implementation is formally verified and automatically translated into OCaml code by Coq. Some representative works of the project are carried out by Blazy et al. [25, 26, 59] and Leroy et al. [92–94].

For instance, Blazy et al. proposed a correctness proof of the translation from a large subset of C language, Clight, into the intermediate language, Cminor, the front-end of the CompCert compiler. The semantic preservation for the translation is formalized as a *simulation* for a Clight program and the translated Cminor program. The semantics of Clight and Cminor, the memory state model, and the simulation are formalized in Coq. A Clight program contains a list of functions, a list of global variables declarations, and the entry point of the program, the *main* function. A Cminor is structured like the Clight language with some differences (e.g., the operators are not overloaded, type casting is explicit,...). The semantics of Clight and Cminor are both specified using big-step operational semantics. We omit the details of evaluation judgements for Clight, Cminor, and the translation. The interested readers should refer to the original article.

To prove the correctness of the translation, the notion of *memory injections* α is introduced to map a block reference b to either `None`, meaning that block has no counter part, or a sub-block b' at offset δ in Cminor memory state. The memory injections are used to define the relation between Clight values v and Cminor v' and the relation between Clight and Cminor memory states, denoted by $\alpha \vdash v \approx v', \alpha \vdash M \approx M'$, respectively. A memory injection α' extends the memory injection α, denoted by $\alpha' \geq \alpha$; it is defined by $\forall b, \alpha'(b) = \alpha(b) \vee \alpha(b) = \texttt{None}$.

A matching relation $EnvMatch(\gamma, \alpha, E, M, E', sp)$ is introduced to match a Clight environment E and memory state M to a Cminor environment E' and reference to a stack

block sp, since the Clight environment E maps local variables to references of blocks containing the values of the variables, while the Cminor environment E' maps directly local variables to the values. γ is a translation environment which reflects the placement of Clight variables. A call stack cs is a list of tuples (γ, E, E', sp). A call stack is *global consistent* with respect to a memory state M and memory injection α, written $CallInv(\alpha, M, cs)$ if $EnvMatch(\gamma, \alpha, E, M, E', sp)$ holds for all elements (γ, E, E', sp) of the stack.

Assuming suitable consistency conditions over the call stack, the semantics of the Clight program is preserved in the translated Cminor program if the generated Cminor expressions and statements evaluate in ways that simulate the evaluation of the corresponding Clight expressions and statements. The definition of this simulation relation is given as follows. Let G' be the global Cminor environment obtained from the global Clight environment G. Assume that $CallInv(\alpha, M, (\gamma, E, E', sp).cs)$ and $\alpha \vdash M \approx M'$. Then, there exists a Cminor environment E'_1, a Cminor memory state M'_1 and a memory injection $\alpha_1 \geq \alpha$ such that

- (R-values) If $G, E \vdash a, M \Rightarrow v, M_1$, $\exists v'$ such that $G', sp, L \vdash \mathscr{R}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash v \approx v'$.
- (L-values) If $G, E \vdash a, M \Rightarrow^l loc, M_1$, $\exists v'$ such that $G', sp, L \vdash \mathscr{L}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash \mathtt{Vptr}(loc) \approx v'$.
- (Statements) If $G, E \vdash s, M \Rightarrow out, M_1$, and τ_r is the return type of the function, $\exists out'$ such that $G', sp \vdash \mathscr{S}_\gamma(s), E', M' \rightarrow out', E'_1, M'_1$ and $\alpha_1, \tau_r \vdash out \approx out'$.

Furthermore, the final memory states M_1 and M'_1 satisfy $CallInv(\alpha_1, M_1, (\gamma, E, E'_1, sp).cs)$ and $\alpha_1 \vdash M_1 \approx M'_1$. The semantic preservation theorem is given as follows. Note that it is assumed that programs always terminate. The semantic preservation is stated as follows: *Assume the* Clight *program* p *is well-typed and translated without errors to a* Cminor *program* p'. *If* $\vdash p \Rightarrow v$, *and if* v *is an integer or float value, then* $\vdash p' \rightarrow v$.

Another example is the work of Leroy [93] which describes the correctness proof of the code generation, the back-end of the CompCert compiler, from a low-level, imperative intermediate language Cminor into optimized PowerPC assembly code, using the Coq proof assistant.

Inspired by the work of CompCert compiler, the formal development of a code generator based on the correct-by-construction components method is carried out in the GeneAuto project [66, 79, 80]. The GeneAuto code generator takes as input a functional description of a system specified in a high-level modeling language (e.g., Simulink, Stateflow) and generates C code as output.

The GENEAUTO toolset is composed of several tools which represent either systems or code models using XML file format. The project focuses on proving the correctness of the *block sequencer* tool which assigns a unique execution order to each block in the system specified in a high-level modeling language. The *block sequencer* has to satisfy the set of *sequencing constraints* which contains:

- Data-flow: blocks computing values used by another block must be executed first.
- Control-flow: caller blocks must be executed during callee blocks after their inputs have been computed.
- Sequential blocks: read part of *Unit Delay* blocks are executed before the write one.
- User priority: two blocks can be sequenced using a user defined priority if they cannot be sequenced with the above constraints.
- Graphical position: if two blocks cannot be sequenced with the above constraints and there is no user defined priority, they are sequenced based on the graphical position defined implicitly in the graphical model.

All the sequencing constraints are translated into formal specification. The specification is written in the functional language GALLINA within the COQ prover. The verification phase proves the conformance to the set of sequencing constraints based on the translation validation approach, meaning that the sequential generated code correctly implements the system specified in a high-level modeling language with respected to the set of sequencing constraints.

Another recent work inspired by the COMPCERT project is the correctness proof of the translation from a small subset of SIGNAL language into the intermediate language, *clock guarded actions* [30, 31], the front-end of a forthcoming verified compiler [138]. The semantics of both input and output of the translation are expressed as *trace semantics*. Then, the formal development of the translation follows a correct-by-construction components method, meaning that for any translation from a primitive operator of SIGNAL into the corresponding clock guarded actions structure, it preserves the trace semantics. The translation is written in the functional programming language GALLINA and formally proved to be correct. The verified translation is translated automatically into OCAML code as the formally verified compiler prototype of the SIGNAL language.

Gamatié et al. [57, 61, 63] introduce an approach to statically analyze SIGNAL programs for efficient code generation. The main idea of their work is that the clocks and clock

relations are formalized as first-order logic formulas with the help of interval-Boolean abstraction technique. This work aims to remove the dead-code segments (e.g., segment of code to compute a data-flow which is always absent). The dead-code segments are identified by detecting the existence of empty clocks, mutual exclusion of two or more clocks, or clock inclusions. The reasoning on the logic formulas is done using a SMT solver. With the interval abstraction, the analysis of clock hierarchy is more precise and more efficient when dealing with the numerical expressions. The common semantics frameworks which are used to construct the translation validation of *clock models* and SDDGs in Chapter 4 and Chapter 5 are based on their interval abstraction technique.

CHAPTER 3

SYNCHRONOUS DATA-FLOW LANGUAGES

This chapter introduces general concepts about reactive systems, the synchronous approach to model reactive systems and the formal verification background. Section 3.1 first defines what reactive systems are. It presents some main features as well as some important issues that designers have to deal with during the design of these systems. Section 3.2 is an introduction to the synchronous approach in designing reactive systems, which has been proposed as a useful approach to describe embedded and safety-critical systems, particularly in automotive and avionics. Then, in Section 3.3, we consider an instance of synchronous programming language, the SIGNAL language. The *polychronous* semantic model which is used to define the formal semantics of the language is studied as well.

3.1 Embedded, reactive and real-time systems

There are several definitions of embedded systems, here, we consider the definition which is proposed by Henzinger and Sifakis [76]. The definition in [60] which conforms to theirs is given as follows:

Definition 1 (Embedded system) *An embedded system is a special-purpose computer system that consists of a combination of software and hardware components that are subject to physical constraints. Such physical constraints come from the system's environment and its execution platform.*

This computer system is embedded as part of a complete device that includes hardware and mechanical parts. Its function is specific, in contrast to a general-purpose computer, such as a personal computer (PC), which is designed to be flexible with a wide range of services.

In general, embedded systems are designed based on *micro-controllers* (i.e CPUs with integrated memory and/or peripheral interfaces). Physically, embedded systems can range

from portable devices such as digital watches, smart phones, to large devices such as factory controllers, hybrid vehicles, and avionics. They can have one micro-controller or multi micro-controllers.

3.1.1 Reactive systems

In [74], Harel and Pnueli introduced the term *"reactive system"*, it is commonly accepted to describe a computer system that continuously interacts with its environment at a speed which is determined by this environment. One example of reactive systems is embedded systems that can be seen anywhere in our modern life. And they are reactive in nature.

Definition 2 (Reactive system) *A reactive system is a computer system that continuously interacts with its environment at a speed which is determined by this environment.*

It is different from a *"transformational system"*, which is a system whose role is to make some outputs computed from some inputs. A transformational system terminates in a finite time duration (e.g., a compiler). It also distinguishes from *interactive* systems, which interact continuously with their environment, but at their own speed (e.g., operating systems).

Reactive systems range from very simple systems (e.g., a system with sensors to record the temperature) to complex systems. Many reactive systems are safety-critical, meaning that even minor errors are unacceptable, and minor errors can make systems go disastrously wrong. For instance, health-related systems, automotive systems, airplane flight control systems and control systems for nuclear plants are reactive systems. The main features of reactive systems which are pointed out in [13, 16, 18, 110] are the following:

Concurrency A reactive system involves concurrency because of the concurrency between the system and its environment. In addition, it is convenient to consider a system as a set of components, which cooperate to achieve the desired behavior. In practice, some systems are implemented on parallel and distributed architectures in order to improve their performance and reliability.

Strict time requirements The systems have to satisfy the requirements about their input rate and their input/output response time. These constraints must be expressed in the system specifications. In order to check these constraints, the evaluation of execution time has especially to be precise during the system design.

Deterministic The outputs of the system are entirely determined by their input values and by the occurrence times of these inputs. The systems will always behave in the same manner, with respect to their expected functional requirements. This makes the validation of the system much easier.

Reliability This can be considered as the most important feature of reactive systems. Errors in reactive systems can be catastrophes, and involve human lives. Therefore, reactive systems require especially rigorous design methods, for instance, formal verification must be considered.

Mixing of hardware and software In many cases, reactive systems are partly implemented by hardware and software, in which they cooperate to achieve the intended function.

3.1.2 Real-time systems

A reactive embedded system has to guarantee a response within a finite and specified time interval, often referred to as "deadline", is called a real-time system. Process control, manufacturing support, command and control are all example application areas where real-time systems have a major role. We consider the definition of a real-time system in [60] which is given as follows:

Definition 3 (Real-time system) *A reactive embedded system is a real-time system when its correctness depends not only on the logical results of its associated computations, but also on the delay after which the results are produced.*

It is common to distinguish between *hard* and *soft* real-time systems. Hard real-time systems are those whose responses occur strictly within the specified deadline. If the deadlines are missed the system will be failed. Soft real-time systems are those where response time is required to be within the specified deadline, however the system still function correctly if deadlines are occasionally missed. Note that most systems combine both hard and soft real-time subparts.

For example, the flight control system of an aircraft is a hard real-time system. A data acquisition is an example of soft real-time system, as it is defined to sample the data from an input sensor at regular time intervals, but it can tolerate some intermittent delays.

Time is obviously a critical resource for real-time systems and must be managed effectively. Unfortunately, it is very difficult to design and implement a system that guarantees

timing requirements under all possible circumstances. Therefore, real-time systems are usually constructed with respect to "worst-case execution time".

In general, real-time programming languages provide the developer real-time control facilities. These control facilities specify times at which actions are to be performed and have to be completed. They also provide the developer solutions to respond to the situations where the timing requirements cannot be met, and the situations where the timing requirements are changed dynamically.

3.2 Synchronous programming

To implement a reactive system, one can use a single loop, of the form in Listing 3.1. This programming scheme is called "event driven" since each reaction is triggered by an input event from the environment.

Listing 3.1 "Event driven"

```
1 < Initialize Memory >
2 for each input_event do
3    < Compute Outputs >
4    < Update Memory >
5 end
```

Listing 3.2 shows a more common programming scheme, which periodically samples the inputs from the environment.

Listing 3.2 "Sampling"

```
1 < Initialize Memory >
2 for each period do
3    < Read Inputs >
4    < Compute Outputs >
5    < Update Memory >
6 end
```

These two programming schemes do not deeply differ, but they correspond to different intuitive points of view. In the following, we shall present the approaches to describe the implementation of the above programming schemes (see [16, 110] for a more complete representation).

3.2.1 Classical approaches

Reactive systems can be modeled using task-based models, finite automata, Petri-net-based models, or classical concurrent and real-time programming languages.

Task-based models The system is designed as a set of sequential tasks. The tasks are activated and controlled by a real-time operating system. The communication between tasks is implemented using a shared memory. The time requirements are guaranteed by means of scheduling instructions (interrupts, priorities,...) and are not directly expressed in the description. Analysis, debug, and maintenance of such systems is hard. In addition, real-time operating systems are generally nondeterministic, which in turn makes programming more difficult.

Finite automata The control kernel of a reactive system is often implemented using an automaton. From a current state, the automaton selects a transition, calls the corresponding sequential tasks, and changes its states for the next reaction. A reaction is neither loop nor recursion with no interrupt and no overhead. The "worst case" execution time can be accurately bounded. Automata are deterministic and can be automatically analyzed by numerous available formal verification techniques [38, 58, 122]. However, writing an automaton with a big number of states is a difficult and error-prone task. Automata, in general, do not directly support hierarchical design and concurrency. Consequently, they are very difficult to use to design complex systems. A small change to the system specifications might involve a complete modification and rewriting of the automaton.

Petri-net-based models Petri-nets can be used to express concurrency in a natural way [124]. But they lack modular structure and often lack determinism. Consequently, it is suitable for small systems.

Classical concurrent and real-time programming languages Concurrent and real-time programming languages such as ADA [4] or OCCAM [96] take concurrency as a primary concern and support modularity. Communication and synchronization mechanisms use rendez-vous, and fifo queues. However, they are essentially asynchronous and nondeterministic: the time taken between the *possibility* of a communication and its actual *achievement* can be arbitrary and is unpredictable. And the order is also unpredictable when several communications take place. In addition, the formal verification of program written in these

languages is often not feasible because asynchrony makes the formal description of the program very large [16].

3.2.2 The synchronous approach

The synchronous approach naturally expresses concurrency. It is deterministic and hierarchical, and possible to use automatic verification tools.

In the synchronous approach [14, 15], time is abstracted by a partial order relation. Events occurring during the same reaction are regarded as simultaneous. Time only increments from one reaction to the next one. Durations between events are not specified. Under this abstraction, computation is considered to take zero time. A synchronous program is supposed to *instantly* and *deterministically* react to the events from its environment. To illustrate the basic idea of synchronous approach, we consider an example in [110] which requires the two following constraints:

- "The train must stop within 10 seconds"
- "The train must stop within 100 meters"

These constraints can be expressed in completely different ways if the physical time is considered. In the synchronous model, they will be expressed by the following constraints:

- "The event *stop* must precede the 10th next occurrence of the event *second*"
- "The event *stop* must precede the 100th next occurrence of the event *meter*"

The notion of *instant* is understood as a logical instant: the history of a system is a totally ordered sequence of logical instants. At each instant, there are zero, one, or several events that can occur. Events which occur at the same instants are considered as *simultaneous*. In the duration between two instants, nothing happens either in the system or its environment. Finally, all the processes of the system have the same knowledge of the events occurring at a given instant.

3.3 The SIGNAL language

3.3.1 Synchronized data flow

We shall consider the concept of synchronized data flow as described in [19, 62, 72]. Based on the data flow semantics given by Kahn [84] as functions over flows, in the following

expression, y is the greatest sequence of values $a'_t + a_t$ where a' is the sub-sequence of strictly positive values in the sequence of values a.

```
1 if a > 0 then
2    x = a
3 y = x + a
```

If we consider an execution where the edges are FIFO queues [8], if a is a finite or infinite sequence of zero or negative values, the queue associated with a grows or grows forever, the queue associated with x is always empty. Assume that the queue consists only of a single cell [48], the execution to compute a value of the sequence associated with y cannot perform as soon as a negative value appears on the input since the first operand does not hold a value (the sequence associated with a is empty), meaning that there exists a deadlock. To deal with this situation in the context of embedded system design, synchronized data flow introduces synchronizations between occurrences of flows. The absent of value is usually represented by *nil*, *null*, or the symbol $\perp$.

3.3.2 An overview of the language

SIGNAL [13, 65] is a polychronous data-flow language that allows the specification of multi-clocked systems. It handles unbounded sequences of *typed* values $x(t), t \in \mathbb{N}$, called *signals*, denoted as x. Each signal is implicitly indexed by a logical *clock* indicating the set of instants at which the signal is present, noted C_x. At a given instant, a signal may be present where it holds a value, or absent where it holds no value (denoted by $\perp$). Given two signals, they are *synchronous* if and only if they have the same clock. In SIGNAL, a process (written as P or Q) consists of the synchronous composition (denoted by $|$) of equations over signals x, y, and z, written as $x := y$ `op` z or $x :=$ `op`(y,z), where `op` is an operator. A program itself is a process.

Data domains Data types contain usual scalar types (Boolean, integer, float, complex, and character), enumerated types, array types, tuple types, and the special type `event`, subtype of the Boolean type which has only one value, `true`.

Operators The core language consists of two kinds of "statements" defined by the following primitive operators: first four operators on signals and last two operators on processes. The operators on signals define basic processes, with implicit clock relations, while the operators on processes are used to construct complex processes with the parallel composition operator.

- *Stepwise functions*: $y := f(x_1,...,x_n)$, where `f` is a *n-ary* function on values, defines the extended stream function over synchronous signals as a basic process whose output y is synchronous with $x_1,...,x_n$ and $\forall t \in C_y, y(t) = f(x_1(t),...,x_n(t))$.

- *Delay*: $y := x\$1$ `init` a defines a basic process such that y and x are synchronous and of $C_x \neq \emptyset$, $y(t_0) = a$, $\forall t \in C_y \wedge t > t_0, y(t) = x(t_-)$, where $t_0 = \inf\{t' | x(t') \neq \bot\}$ and $t_- = \sup\{t' | t' < t \wedge x(t') \neq \bot\}$.

- *Merge*: $y := x$ `default` z defines a basic process which specifies that y is present if and only if x or z is present, and that $y(t) = x(t)$ if $t \in C_x$ and $y(t) = z(t)$ if $t \in C_z \setminus C_x$.

- *Sampling*: $y := x$ `when` b where b is a Boolean signal, defines a basic process such that $\forall t \in C_x \cap C_b \wedge b(t) = \texttt{true}, y(t) = x(t)$, and otherwise, y is absent.

- *Composition*: If P_1 and P_2 are processes, then $P_1 \mid P_2$, also denoted $(|P_1 \mid P_2|)$, is the process resulting of their parallel composition. This process consists of the composition of the systems of equations. The composition operator is commutative, associative, and idempotent.

- *Restriction*: P/x, where P is a process and x is a signal, specifies a process by considering x as local variable to P (i.e., x is not accessible from outside P).

Clock relations In addition, the language allows clock constraints to be defined explicitly by some derived operators that can be replaced by primitive operators above.

- *Clock extraction*: $y := \hat{x}$ specifies that y is the clock of x with type `event`. It is equivalent to $y := (x = x)$ in the core language.

- *Synchronization*: $x \hat{=} y$ means that x and y have the same clock. It can be replaced by $\hat{x} = \hat{y}$.

- *Clock extraction from Boolean signal*: `when` b indicates the sub-clock $[b]$. It is the shortcut for b `when` b.

- *Clock union*: $x \hat{+} y$ defines a clock as the union $C_x \cup C_y$, which can be rewritten as $\hat{x}$ `default` $\hat{y}$.

- *Clock intersection*: $x \hat{*} y$ defines a clock as the intersection $C_x \cap C_y$, which can be rewritten as $\hat{x}$ `when` $\hat{y}$.

	Dependency	Clock relation
x	$C_x \xrightarrow{C_x} x$	
c (Boolean signal)	$c \xrightarrow{[c]} [c]$, $c \xrightarrow{[\neg c]} [\neg c]$	
$x \xrightarrow{c} y$	$[c] \xrightarrow{[c]} y$	
$y := f(x_1,...,x_n)$	$x_1 \xrightarrow{C_y} y \ ... \ x_n \xrightarrow{C_y} y$	$C_y = C_{x_1} \ ... \ C_y = C_{x_n}$
$y := x\$1$ init a		$C_y = C_x$
$y := x$ when b	$x \xrightarrow{C_y} y$, $b \xrightarrow{C_y} C_y$	$C_y = C_x \cap [b]$
$y := x$ default z	$x \xrightarrow{C_x} y$, $z \xrightarrow{C_z \setminus C_x} y$	$C_y = C_x \cup C_z$

Table 3.1 The implicit clock relations and dependencies

- *Clock difference*: $x \hat{-} y$ defines a clock as the set $C_x \setminus C_y$, which can be rewritten as `when` (`not`($\hat{y}$) `default` $\hat{x}$).

Implicit clock relations and dependencies The above basic processes induce clock relations and dependencies among signals. Table 3.1 shows these clock constraints for the primitive operators. In this table, the sub-clock $[c]$ (resp. $[\neg c]$) is defined as $\{t \in C_c | c(t) = \texttt{true}\}$ (resp. $\{t \in C_c | c(t) = \texttt{false}\}$). Notice that a clock can be viewed as a signal with type `event` (which has only one value, `true`, when it is present), thus the condition C_c means that the signal c is present.

Let x, y be two signals and c be a Boolean signal, if at any instant t such that $t \in C_x \cap C_y \cap C_c$ and $c(t) = true$, setting a value to y cannot precede the availability of x, then we say that y depends on x at the condition c. We use $x \xrightarrow{c} y$ to denote the fact that there is a dependency between y and x at the condition c. In particular, the following dependencies apply implicitly.

- Any signal is preceded by its clock.
- For a Boolean signal c, $[c]$ and $[\neg c]$ depend on c.
- Any dependency $x \xrightarrow{c} y$ implies implicitly a dependency $[c] \xrightarrow{[c]} y$.

Example The following Signal program emits a sequence of values $FB, FB-1, ..., 2, 1$, from each value of a positive integer signal FB coming from its environment.

Listing 3.3 DEC in Signal

```
1 process DEC=
2 (? integer FB;
3  ! integer N)
4  (| FB ^= when (ZN<=1)
5   | N := FB default (ZN-1)
6   | ZN := N$1 init 1
7   |)
8 where integer ZN
9 end;
```

Let us comment this program. FB, and N are respectively input and output signals of type `integer` (line 2 and 3). FB is accepted, it is present, only when ZN becomes less than or equal to 1 (line 4). N is set to FB when its previous value is less than or equal to 1, otherwise it is decremented by 1 (line 5). ZN is defined as carrying the previous value of N and its initial value is 1) (line 6). ZN is a local signal (line 8).
We can see that the clock of the output signal is more frequent than that of the input. This is illustrated in the following possible traces:

```
1  t  .   .   .   .   .   .   .   .   .   .
2 FB  6   ⊥   ⊥   ⊥   ⊥   ⊥   3   ⊥   ⊥   2
3 ZN  1   6   5   4   3   2   1   3   2   1
4  N  6   5   4   3   2   1   3   2   1   2
5 C_FB t_0                     t_6         t_9
6 C_ZN t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
7 C_N  t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
```

Program structure The language is modular, meaning that a process can be used as a basic pattern, by means of an interface that describes its parameters and its input and output signals. Moreover, a process can use other subprocesses, or even external parameter processes that are only known by their interfaces. For example, to emit three sequences of values $FB_i - 1, \ldots, 2, 1$ for all three positive integer inputs FB_i, with $i = 1, 2, 3$, one can define the following process (in which, without additional synchronizations, the three subprocesses have unrelated clocks):

```
1 process 3DEC=
2 (? integer FB1, FB2, FB3;
3  ! integer N1, N2, N3)
4  (| N1 := DEC(FB1)
5   | N1 := DEC(FB2)
```

```
6    | N3 := DEC(FB3)
7    |)
8 end;
```

3.3.3 Semantics of the language

This section presents the operational and denotational semantics of the SIGNAL language. Operational semantics are classified in two categories: *big-step semantics* and *small-step semantics* [116]. Big-step semantics describe how the overall results of the executions are obtained. By opposition small-step semantics formally describe how the individual steps of a computation take place on an abstract machine. Denotational semantics [127] formalize the meanings of programming languages by constructing mathematical objects (called *denotations*) that describe the meanings of expressions from the languages. Denotational semantics describe a valid program as a function from its inputs to its outputs. The operational and denotational semantics of SIGNAL were originally proposed by Benveniste et al. in [14] and Le Guernic et al. in [72]. They are also presented by Gamatié in the book on designing embedded systems with the SIGNAL language [60].

3.3.3.1 Operational semantics

Let X and $\mathbb{V}$ be a countable set of variables and their domain of values, respectively. The domain with the absent value is denoted by $\mathbb{V}^{\perp} = \mathbb{V} \cup \perp$. A signal variable can be assigned values according to an *environment* that is defined as follows:

Definition 4 (Environment) *Let $X_1 \subset X$ be a set of signal variables, an environment associated with X_1 is defined as a function $\varepsilon : X_1 \rightarrow \mathbb{V}^{\perp}$. This function assigns values to signals at each instant during an execution.*

The set of environments associated with X_1 is written as ε_{X_1}. When the signal s is present, the environment assigns a value $v \in \mathbb{V}$ to s (denoted by $s(v) \in \varepsilon$, or $\varepsilon(s) = v$). Otherwise, s is associated with the special value, absent value (noted as $s(\perp)$). An environment where all signals are absent is denoted by $\perp_{\varepsilon}$.

Definition 5 (Environment restriction) *Consider an environment ε associated with X and a subset $X_1 \subseteq X$, the restriction written as $\varepsilon_{\overline{X_1}}$ is defined as follows:*

$$\varepsilon_{\overline{X_1}} : X \setminus X_1 \rightarrow \mathbb{V}^{\perp} \text{ such that } \forall x \in X \setminus X_1, \varepsilon(x) = \varepsilon_{\overline{X_1}}(x)$$

For example, we consider the following environment:

$$\varepsilon : \{s_1, s_2, s_3\} \to \{v_1, v_2, v_3, v_4\}; \varepsilon(s_1) = v_1, \varepsilon(s_2) = v_2, \varepsilon(s_3) = v_3$$

Then, the environment restriction $\varepsilon_{\overline{s_2}}$ is defined as:

$$\varepsilon_{\overline{s_2}} : \{s_1, s_3\} \to \{v_1, v_2, v_3, v_4\}; \varepsilon(s_1) = v_1, \varepsilon(s_3) = v_3$$

The signal s_2 is not visible in the environment restriction.

Definition 6 (Environment composability) *Let $\varepsilon_1 \in \varepsilon_{X_1}$ and $\varepsilon_2 \in \varepsilon_{X_2}$ be two environments. If for all signal variables $x \in X_1 \cap X_2$, $\varepsilon_1(x) = \varepsilon_2(x)$, then they are composable and their composition $\varepsilon_1 \oplus \varepsilon_2$ is defined as follows:*

$$\begin{aligned} \oplus : & \varepsilon_{X_1} \times \varepsilon_{X_2} \to \varepsilon_{X_1 \cup X_2} \\ & (\varepsilon_1, \varepsilon_2) \mapsto \varepsilon_1 \cup \varepsilon_2 \end{aligned}$$

For example, we consider the following environments:

$$\begin{aligned} \varepsilon_1 : & \{s_1, s_2, s_3\} \to \{v_1, v_2, v_3, v_4\}; \varepsilon(s_1) = v_1, \varepsilon(s_2) = v_2, \varepsilon(s_3) = v_3 \\ \varepsilon_2 : & \{s_1, s_4, s_5, s_6\} \to \{v_1, v_4, v_5, v_6\}; \varepsilon(s_1) = v_1, \varepsilon(s_4) = v_4, \varepsilon(s_5) = v_5, \varepsilon(s_6) = v_6 \end{aligned}$$

Since ε_1 and ε_2 are composable, we have their composition environment as follows:

$$\begin{aligned} \varepsilon_1 \oplus \varepsilon_2 : & \{s_1, s_2, s_3, s_4, s_5, s_6\} \to \{v_1, v_2, v_3, v_4, v_5, v_6\}; \\ & \varepsilon(s_1) = v_1, \varepsilon(s_2) = v_2, \varepsilon(s_3) = v_3, \varepsilon(s_4) = v_4, \varepsilon(s_5) = v_5, \varepsilon(s_6) = v_6 \end{aligned}$$

The operational semantics of the SIGNAL language is described by a labeled transition systems where the states are SIGNAL processes. It is given as follows:

$$\frac{C}{p_1 \xrightarrow{\varepsilon} p_2}$$

where p_1 and p_2 are processes, ε is an execution environment, and C is a *precondition* on p_1, p_2, and ε. We now define the operational semantics for all primitive operators of the language SIGNAL. Consider the environment in which all signals involved in a process p are absent. Hence, the process p never reacts. The operational semantics of this process can

be represented by the following *trivial rule*:

$$p \xrightarrow{\perp_\varepsilon} p$$

Stepwise functions Let p be the basic process $y := f(x_1,...,x_n)$, the signal y is present and holds the value calculated by the expression $f(v_1,...,v_n)$ when the signal $x_1,...,x_n$ are present and carry the values $v_1,...,v_n$, respectively. The operational semantics of p is defined as follows:

$$p \xrightarrow{x_1(v_1),...,x_n(v_n),y(f(v_1,...,v_n))} p$$

When all signals are absent, the operational semantics is represented by the trivial rule.

Delay The operational semantics of the basic process $p = y := x\$1$ `init` a is defined by the following transition rule.

$$p \xrightarrow{x(v),y(a)} p'$$

where $p' = y := x\$1$ `init` v. The *delay* operator expresses a dynamic behavior in which the value of the signal x is memorized after each transition. The process p' memorizes the previous value v carried by x as the new initialization value for the next transition. The trivial rule is also applied for the *delay* operator.

Merge The semantics of *merge* operator, $y := x$ `default` z, is defined by the following transition rules:

$$p \xrightarrow{x(v_1),z(\perp),y(v_1)} p$$
$$p \xrightarrow{x(v_1),z(v_2),y(v_1)} p$$
$$p \xrightarrow{x(\perp),z(v_2),y(v_2)} p$$
$$p \xrightarrow{\perp_\varepsilon} p$$

Sampling The semantics of *sampling* operator, $y := x$ `when` b, is defined by the following transition rules:

$$p \xrightarrow{x(v),b(\bot),y(\bot)} p$$
$$p \xrightarrow{x(\bot),b(v_b),y(\bot)} p$$
$$p \xrightarrow{x(v),b(\texttt{true}),y(v)} p$$
$$p \xrightarrow{x(v),b(\texttt{false}),y(\bot)} p$$
$$p \xrightarrow{\bot_\varepsilon} p$$

Composition Let p_1, p_2, p_3, and p_4 be four processes within the environments ε_1 and ε_2. The operational semantics of the composition operator is defined as follows:

$$\frac{p_1 \xrightarrow{\varepsilon_1} p_2; p_3 \xrightarrow{\varepsilon_2} p_4; \varepsilon_1 \text{ and } \varepsilon_2 \text{ are composable}}{(p_1 \mid p_3 \xrightarrow{\varepsilon_1 \oplus \varepsilon_2} (p_2 \mid p_4)}$$

The upper part of the transition rule defines the condition at which the composition of two processes is possible (the two environments must be composable). The lower part describes the composition whenever this condition is satisfied.

Restriction Let p_1 and p_2 be two processes within an environment ε. The semantics of the process restriction is simply defined by restricting the visibility of the signals concerned in the transition environment. The operational semantics is defined as follows:

$$\frac{p_1 \xrightarrow{\varepsilon} p_2}{p_1/x \xrightarrow{\varepsilon_{\overline{x}}} p_2/x}$$

3.3.3.2 Denotational semantics

Let $X, \mathbb{B} = \{ff, tt\}, \mathbb{V} \supseteq \mathbb{B}$, and $\mathbb{T}$ be a countable set of variables, a set of Boolean values (ff and tt denote `false` and `true`, respectively), the domain of variables, and a dense set equipped with a partial order relation $\leq$. The elements in $\mathbb{T}$ are called *tags*. The partially ordered set $(\mathscr{T}, \leq)$ is a subset $\mathscr{T} \subset \mathbb{T}$ such that:

- $\mathscr{T}$ is countable,
- $\mathscr{T}$ has a lower bound 0 for $\leq$,

- the partial order $\leq$ on $\mathscr{T}$ is well-founded.

We denote by $\mathscr{C}_{\mathscr{T}}$ the set of all chains C in $\mathscr{T}$. Then, for a tag $t \in C$, min(C), max(C) and $\text{pred}_C(t)$ denote the minimum, maximum and the predecessor of t.

Definition 7 (Event, signal and behavior) *Given a set $\mathscr{T}$, an event $e \in E$ is a relation between a tag and a value. A signal $s \in \mathscr{S} = \mathscr{T} \rightharpoonup \mathbb{V}$ is a partial function defined on a chain of tags to a set of values. A behavior $b \in \mathscr{B} = X \rightharpoonup \mathscr{S}$ is a partial function that maps a signal name $x \in X$ to a signal $s \in \mathscr{S}$.*

We write tags(s) and vars(b) to denote the domain of s and domain of b, respectively. Then, $\text{tags}(b) = \bigcup_{x \in vars(b)} tags(b(x))$ denote the tags of the behavior b. Therefore, x is present at instant t in behavior b can be formally expressed by $t \in tags(b(x))$.

For example, consider the following trace, we have:

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	...
x_1 :	1				2		3				...
x_2 :		1				2		3			...
x_3 :			1	2					3		...

- $X = \{x_1, x_2, x_3\}$
- $\mathbb{V} = \{1, 2, 3\}$
- $\mathscr{T} = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, ...\}$
- $\mathscr{E} = \{e_0 = (t_0, 1), e_1 = (t_1, 1), e_2 = (t_2, 1), e_3 = (t_3, 2), e_4 = (t_4, 2), e_5 = (t_5, 2), e_6 = (t_6, 3), e_7 = (t_7, 3), e_8 = (t_8, 3), ...\}$
- $\mathscr{S} = \{s_1 = \{e_0, e_4, e_6, ...\}, s_2 = \{e_1, e_5, e_7, ...\}, s_3 = \{e_2, e_3, e_8, ...\}\}$
- $\mathscr{B} = \{(v_1, s_1), (v_2, s_2), (v_3, s_3)\}$

The projection of a behavior b on a set $X_1 \subset X$ of names, $b\,|_{X_1}$, is defined as follows:

- $\text{vars}(b\,|_{X_1}) = X_1$
- $\forall x \in X_1, b\,|_{X_1}\,(x) = b(x)$

We write $b_{/X_1}$ for the projection of b on the complementary of X_1 in vars(b). The empty signal is denoted by λ. $0\,|_{X_1} = \{(x, \lambda) \mid x \in X_1$ to denote the association of $X_1 \subset X$ to the empty signal.

Definition 8 (Process) *A process $p \in \mathscr{P}$ is a stretch-closed set of behaviors which have the same domain X (denoted by vars(p)). The synchronous composition of two processes $p \mid q$ is defined by the set of behaviors that extend a behavior $b \in p$ by the restriction $c_{/vars(p)}$ of a behavior $c \in q$ provided that the projections of b and c on $vars(p) \cap vars(q)$ are equal.*

$$p \mid q = \{b \uplus c_{/vars(p)} \mid (b,c) \in p \times q, b \mid_{vars(p)\cap vars(q)} = c \mid_{vars(p)\cap vars(q)}\}$$

Definition 9 (Stretching) *A behavior c is a stretching of b, noted $b \leq_{\mathscr{B}} c$, iff vars(b) = vars(c) and there exists a bijection $f : \mathscr{T} \to \mathscr{T}$ such that:*

- $\forall t,u \in \mathscr{T}, t \leq u \leftrightarrow f(t) \leq f(u)$
- $\forall C \in \mathscr{C}, \forall t \in C, t \leq f(t)$
- $forall x \in vars(b), tags(c(x)) = f(tags(b(x)))$
- $\forall x \in vars(b), \forall t \in tags(b(x)), b(x)(t) = c(x)(f(t))$

The stretching relation is a partial order on behaviors $\mathscr{B}$. This provides an equivalence relation between two behaviors.

Definition 10 (Stretch equivalence) *The behaviors b and c are stretch-equivalent, written $b \lessgtr c$, iff there exists a behavior d such that $d \leq_{\mathscr{B}} b$ and $d \leq_{\mathscr{B}} c$.*

For a behavior b, the class of all behaviors that are stretch-equivalent is denoted by b^*. The stretch closure of a set of behaviors p, denoted by p^*, is defined as $p^* = \bigcup_{b\in p} b^*$. In particular, it is interesting to consider the class of processes which contain all possible stretches of a given behavior.

Definition 11 (Stretch closure) *A process p is stretching-closed iff for all $b \in p$ and all $c \in \mathscr{B}, c \lessgtr b \Rightarrow c \in p$.*

A stretching-closed process p admits a set of strict behaviors, denoted $(p)_{\lessgtr}$, such that $(p)_{\lessgtr} \subset p$. It satisfies that for all behavior $b \in p$, there exists a unique behavior $c \in (p)_{\lessgtr}$ such that $c \lessgtr b$.

For each primitive operator of SIGNAL, we describe its denotational semantics in terms of sets of behaviors. We denote by $[[p]]$ the set of all possible behaviors of the process p.

Stepwise functions The denotational semantics of the *stepwise functions*, $y := f(x_1, ..., x_n)$, is defined as follows:

$$\begin{aligned}[[y := f(x_1, ..., x_n)]] = \{b \in \mathscr{B} \mid_{y,x_1,...,x_n} | \\ tags(b(y)) = tags(b(x_1)) = ... = tags(b(x_n)) = C \in \mathscr{C} \setminus \emptyset \\ \forall t \in C, b(y)(t) = [[f]](b(x_1)(t), ..., b(x_n)(t))\}\end{aligned}$$

The denotational semantics of the *stepwise functions* is the set of behaviors b such that:

- The tags of each signal x_i represent the same chain C of tags.
- For each tag in C, the relation f holds between the values carried by the involved signals. We denote the stepwise interpretation of f by $[[f]]$.

Delay The denotational semantics of the *delay* operator, $y := x\$1$ `init` a, is defined as follows:

$$\begin{aligned}[[y := x\$1 \texttt{ init } a]] = \{0 \mid_{y,x}\} \cup \{b \in \mathscr{B} \mid_{y,x} | \\ tags(b(y)) = tags(b(x)) = C \in \mathscr{C} \setminus \emptyset \\ b(y)(min(C)) = a, \forall t \in C \setminus min(C), b(y)(t) = b(x)(pred_C(t))\}\end{aligned}$$

The denotational semantics of the *delay* operator is the set of behaviors b such that:

- The tags of the signals y and x represent the same chain C of tags.
- At the initial tag of C, y holds the value a.
- For all other tag t in C, y holds the value that is carried by x at the predecessor of t.

Merge The denotational semantics of the *merge* operator, $y := x$ `default` z, is defined as follows:

$$\begin{aligned}[[y := x \texttt{ default } z]] = \{b \in \mathscr{B} \mid_{y,x,z} | \\ tags(b(y)) = tags(b(x)) \cup tags(b(z)) = C \in \mathscr{C} \\ \forall t \in C, b(y)(t) = b(x)(t) \text{ if } t \in tags(b(x)) \text{ else } b(z)(t)\}\end{aligned}$$

The set of behaviors satisfy:

- The tags of the signals y is the union of those associated with x and z.

- The value taken by y is the value carried by x at any common tag of x and y. Otherwise, y takes the value carried by z.

Sampling The denotational semantics of the *sampling* operator, $y := x$ `when` b, is defined as follows:

$$\begin{aligned}[[y := x \texttt{ when } b]] = \{b \in \mathscr{B} \mid_{y,x,b} \mid \\ tags(b(y)) = \{t \in tags(b(x)) \cap tags(b(b)) \mid b(b)(t) = tt\} = C \in \mathscr{C} \\ \forall t \in C, b(y)(t) = b(x)(t)\}\end{aligned}$$

The denotational semantics of the *sampling* operator consists of the behaviors which satisfy:

- The tags of the signals y is the intersection of the tags of x and the set of tags at which b carries the value `true`.
- At each tag of y, it holds the value carried by x.

Composition The composition of two processes p and q is the stretch closure of the behaviors b such that the projections of these behaviors on vars(p) and on vars(q) give behaviors in p and q, respectively. The denotational semantics is given as follows:

$$[[p \mid q]] = (\{b \mid b \mid_{vars(p)} \in p, b \mid_{vars(q)} \in q\})^*$$

Restriction The restriction of a process p over x is the stretch closure of the behaviors of p projected on the complementary set of $\{s\}$. The denotational semantics is defined as follows:

$$[[p/x]] = (\{c \mid \exists b \in p \wedge c \leqslant b_{/\{x\}}\})^*$$

CHAPTER 4

TRANSLATION VALIDATION OF TRANSFORMATIONS ON CLOCKS

Clocks and clock relations are used to represent all the control parts (e.g. activation events) and interaction between different components in a system. The control flow resulting from the analysis of clocks and clock relations is used to derive an optimized data-flow following the transformations of the compiler. Therefore, the correctness of clock analysis in the compilation of synchronous programs strongly impacts the quality of the compiled program.

In this chapter, we describe how the preservation of clock semantics can be proved based on the translation validation approach. It focuses on proving that for any transformation in the first phase of the SIGNAL compilation process, the source program and its compiled program have the same clock semantics. We propose two approaches based on model checking and the use of SMT solver.

In the first approach, the clock semantics of the programs are represented by *Polynomial Dynamical Systems* (PDSs) that is considered as a common semantic framework in our translation validation. A PDS *refinement* is formally formalized to express the concept of correct transformation and provide a method to prove the existence of this refinement based on the simulation technique.

In the second approach, the presence/absence status of all signals in a program at a given instant is deterministically characterized by a first-order logic formula with *uninterpreted functions*, called *clock model*. Given two clock models, a *clock refinement* between them is defined which expresses the preservation of clock semantics. A method to check the existence of clock refinement is defined as a first-order logic formula satisfiability problem.

We design a validator and plug it within the *clock calculation* and *Boolean abstraction* phase. Figure 4.1 shows the integration of this validator into the compilation process. The validator takes the source program and its compiled counterpart, and constructs the corre-

sponding PDSs or clock models for the first and second approaches, respectively. It then checks the existence of PDS refinement or clock refinement depending on which approach is being used. If the result is that the relation does not exist then a "compiler bug" message is emitted. Otherwise, the compiler continues its work.

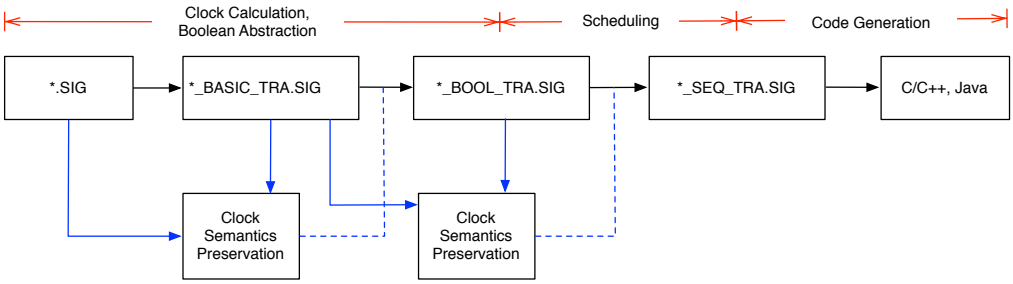

Fig. 4.1 A bird's-eye view of the verification process

The remainder of this chapter is organized as follows. Section 4.1, first, describes how the SIGNAL compiler calculates the clocks and makes the Boolean abstraction of synchronous programs to answer some important questions in the first phase compilation. Section 4.2 addresses the $\mathbb{Z}/_3\mathbb{Z}$ encoding of signal clocks and values. In Section 4.3, we consider the formal definition of PDSs, the translation of SIGNAL programs into PDSs and translation validation of PDSs. Section 4.4 provides the definition of clock model, the encoding scheme from SIGNAL programs into clock models and the translation validation of clock models. We also address the mechanism of the verification process, the application of the verification process to the SIGNAL compiler, and its integration in the POLYCHRONY toolset [77]. Section 4.5 presents some related works, concludes our work and outlines some future directions.

4.1 The clock calculus in SIGNAL compiler

Consider the first phase of the SIGNAL compilation process, the goal of this phase is calculating clocks of all signals, synthesizing clock relations, and providing a Boolean abstraction of a program. The compiled program is also written in SIGNAL language as depicted Figure 1.4. The compiler represents the clocks and clock relations as a system of equations. These equations are encoded in the finite field $\mathbb{Z}/_3\mathbb{Z}$ of integers modulo 3, where -1 encodes the presence of a Boolean signal holding the value `false`, 0 encodes the absence of a signal, $+1$ encodes the presence of a Boolean signal holding the value `true`. The full encoding

scheme of SIGNAL programs will be presented in Section 4.2 and Section 4.2.1.

The analysis of such a system of equations provides some important functions of the compiler [13]. The first function of this phase is the detection of empty behavior and synchronization errors. If the system of equations has only one solution such that the empty clock is set to every signal, then the program cannot execute. For instance, we consider the following program:

```
1 (| x := a when (a > 0)
2  | y := a when not (a < 0)
3  | z := x + y
4  |)
```

which induces $[a > 0] = [\neg(a > 0)] = C_a \setminus [a > 0]$ (since x and y are synchronous), because $[a > 0] \subseteq C_a$, we get $C_a = C_x = C_y = C_z = \emptyset$.

If the program can be executed but some inputs from the environment are not accepted, then the program can be deadlocked. This happens in the following program:

```
1 (| c := a > 0
2  | x := a when c
3  | y := x + a
4  |)
```

which provides $C_a \cap [c] = C_a$ and $C_a = C_c$, implies that $[c] = C_c$, which means that the input signal a must always be positive. In consequence of this constraint on the input, this program will be rejected if its environment provides a negative value for a which implies that the signal a is present, c is present and holds the value `false` and x is absent.

The other function of this phase is to construct the control flow of the program. Given a program, it may be established an order relation on the set of clocks: the clock of signal h is said to be greater than the clock of signal k if the set of instants of k is a sub-set of the set of instants of h (k is an under-sampling of h), denoted by $C_h \geq C_k$. Equivalently, we say that the class C_h dominates the class C_k. The relation $\geq$ on the quotient set of signals by $\hat{}=$ (two signal are in the same class if and only if they are synchronous) is called a *clock hierarchy* [7]. When the clock hierarchy has a unique highest class, then the program has a fastest rated clock (called the *master clock*) and the definitions of other clocks are expressed as under-samplings of the fasted rated clock. In this case, the status (presence/absence) of all signals is a pure flow function: this status depends neither on communication delays, nor on computing latencies.

A program which contains only one master clock is referred to as *endochronous*, meaning that there is a unique way to compute the clocks of its signals. Endochronous programs

can be executed in a *deterministic* way [72]. If there exists more than one master clock, some parts of the program can run at independent rates. Consider the following example:

```
1 (| x := (x$ init 0) + 1
2  | y := x when c
3  |)/x
```

We get the clock relation $C_y = C_x \cup [c]$, which leaves the clock of x and c unrelated. The signal x is not synchronized by any external signal, thus its computation rate is undetermined. As a result, the output y can be any subsequence of the sequence of integers. To make the program execute in a deterministic way, this program needs extra information from its environment (i.e., the clock of x needs to be synchronized by some external signal).

4.2 The synchronization space $\mathbb{Z}/_3\mathbb{Z}$

We denote by $\mathbb{Z}/_p\mathbb{Z}[Z]$ the set of polynomials over variables $Z = \{z_1, ..., z_k\}$ whose coefficients range over the finite field modulo p, $\mathbb{Z}/_p\mathbb{Z}$, with p prime. For a polynomial $P \in \mathbb{Z}/_p\mathbb{Z}[Z]$, the solutions of the equation $P(Z) = 0$ are denoted by $Sol(P)$.

We say that $P_1 \equiv P_2$ whenever $Sol(P_1) = Sol(P_2)$. And the representative of $Sol(P)$ of each $\equiv$-equivalence class is called the *canonical generator*. In the rest of this chapter, we shall use the following notations, where $P|_{z_i=v}$ is P obtained by instantiating any occurrence of variable z_i by value v.

$$
\begin{array}{lcl}
\overline{P} & \triangleq & 1 - P^{p-1} \\
P_1 \oplus P_2 & \triangleq & (P_1^{p-1} + P_2^{p-1})^{p-1} \\
P_1 \Rightarrow P_2 & \triangleq & \{Z \in (\mathbb{Z}/_p\mathbb{Z})^k | P_1(Z) = 0 \Rightarrow P_2(Z) = 0\} \\
\exists z_i P & \triangleq & P|_{z_i=1} * P|_{z_i=2} * ... * P|_{z_i=p} \\
\forall z_i P & \triangleq & P|_{z_i=1} \oplus P|_{z_i=2} \oplus ... \oplus P|_{z_i=p}
\end{array}
$$

Following the above notations, we can see that the solutions of $\overline{P}$ are defined as $Sol(\overline{P}) = (\mathbb{Z}/_p\mathbb{Z})^k \backslash Sol(P)$. And the polynomial $P_1 \Rightarrow P_2$ has the same the set of solutions as $\overline{P_1} * P_2$, or $P_1 \Rightarrow P_2 \equiv \overline{P_1} * P_2$. The manipulations of polynomials over the finite field modulo p, with p prime can be found in [29].

We consider the finite field modulo 3, $\mathbb{Z}/_3\mathbb{Z} = \{-1, 0, +1\}$. This field holds some very interesting properties that are used to reason about properties of encoded statements. For

instance, let x be an element in $\mathbb{Z}/_3\mathbb{Z}$ and $n \in \mathbb{N}$, then:

$$\begin{array}{lcl} x^{2n} & = & x^2 \\ x^{2n+1} & = & x \\ x+x & = & -x \\ \frac{1}{x} & = & x, \forall x \neq 0 \\ x(1-x^2) & = & 0 \\ (f(x^2))^{2n} & = & f(x^2) \end{array}$$

Our aim is to build a formal model which represents the behaviors of synchronous data-flow programs in terms of the presence of signals and the relations among signals. Let us, by the way of illustration, consider the basic SIGNAL process corresponding to the primitive sampling operator, $y := x$ `when` b, where x and y are non-Boolean signals and b is a Boolean signal. This basic process expresses the following statements:

- if x is present, and b is present and holds the value `true`, then y is defined by the value of x,
- if x is absent, or b is absent, or b is present and holds the value `false`, then y is absent.

It shows that at a given instant, to express the control, we need to represent the status of the signals x, y and b. In this example, for the non-Boolean signals such as x, its status are: x is present and x is absent. For the Boolean signal b, its status are: b is present and holds the value `false`, b is present and holds the value `true` and b is absent. The principle is that the status of a signal can be encoded in the finite field modulo $p = 3, \mathbb{Z}/_3\mathbb{Z}$. This encoding scheme were proposed by Le Guernic et al. in [62, 72] as a model of the synchronization relation. For a Boolean signal, its three possible status at a given instant are encoded as:

- `present` and `false` $\rightarrow -1$,
- `present` and `true` $\rightarrow 1$,
- `absent` $\rightarrow 0$.

For a non-Boolean signal, it only encodes the fact that the value is present or absent (it does not encode value):

- `present` $\rightarrow \pm 1$,
- `absent` $\rightarrow 0$.

In this way, if x is the encoding of the signal x, its clock can be considered as $x^2 : 1$ if x is present; 0 if x is absent. Thus, for two synchronous signals x and y (they have the same clock), x and y satisfy the constraint equation: $x^2 = y^2$.

4.2.1 PDS model

Processes are translated into systems of polynomial equations over the finite field modulo $p = 3$, $\mathbb{Z}/_3\mathbb{Z}$, following the above encoding scheme. Each individual SIGNAL equation is translated into a polynomial equation. Since the language is defined from a few primitive operators, we only need to define the translation of these primitive operators to polynomial equations.

4.2.1.1 Stepwise functions

The functions which apply on signal values in the primitive *stepwise functions* are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`), and numerical operators (`+`, `-`, `*`, `/`). The encoding in the finite field modulo $p = 3$, $\mathbb{Z}/_3\mathbb{Z}$, of stepwise functions is given as follows:

- If x and y are Boolean signals:

$$
\begin{array}{lcl}
y := \texttt{not}\ x & \longrightarrow & y = -x \\
z := x\ \texttt{and}\ y & \longrightarrow & \left\{ \begin{array}{l} z = xy(xy - x - y - 1) \\ x^2 = y^2 \end{array} \right. \\
z := x\ \texttt{or}\ y & \longrightarrow & \left\{ \begin{array}{l} z = xy(1 - x - y - xy) \\ x^2 = y^2 \end{array} \right.
\end{array}
$$

- If x and y are non-Boolean signals:

$$y := f(x_1, ..., x_n) \longrightarrow y^2 = x_1^2 = ... = x_n^2$$

4.2.1.2 Delay

Considering the *delay* operator, $y := x\$1\ \texttt{init}\ a$, its encoding is given as follows:

- If x, y and a are Boolean signals:

$$y := x\$1 \texttt{ init } a \quad \longrightarrow \quad \begin{cases} m.x' = x + (1 - x^2)m.x \\ y = x^2 m.x \\ m.x = a \end{cases}$$

- If x, y and a are non-Boolean signals:

$$y := x\$1 \texttt{ init } a \quad \longrightarrow \quad y^2 = x^2$$

This encoding requires that at any instant, signals x and y have the same status (present or absent). If the signals are Boolean, it encodes the value of the output signal as well. Here, we introduce a memorization variable $m.x$ that stores the last value of x. The next value of $m.x$ is $m.x'$ and it is initialized to a in $m.x = a$.

4.2.1.3 Merge

The encoding of the *merge* operator, $y := x \texttt{ default } z$ is given as follows:

- If x, y and z are Boolean signals:

$$z := x \texttt{ default } y \quad \longrightarrow \quad z = x + (1 - x^2)y$$

- If x, y and z are non-Boolean signals:

$$z := x \texttt{ default } y \quad \longrightarrow \quad z^2 = x^2 + y^2 - x^2 y^2$$

For Boolean signals, the encoding equation can be interpreted as follows: if x is defined then z is defined and holds the same value as x ($z = x + (1 - 1)y = x$). Otherwise, when y is defined and x is not, then z holds the same value as y ($z = 0 + (1 - 0)y = y$).

4.2.1.4 Sampling

The encoding of the *sampling* operator, $y := x \texttt{ when } b$, is given as follows:

- If x, y and z are Boolean signals:

$$y := x \texttt{ when } b \quad \longrightarrow \quad y = x(-b - b^2)$$

- If x, y and z are non-Boolean signals:

$$y := x \texttt{ when } b \quad \longrightarrow \quad y^2 = x^2(-b-b^2)$$

The encoding of this operator for Boolean signals may be interpreted as follows: y holds the same value as x when b is defined and holds the value `true` ($y = x(-1-1) = x$).

4.2.1.5 Composition

Consider the composition of two processes P_1 and P_2. The encoding of $\phi(P_1|P_2)$ in the finite field modulo $p = 3$, $\mathbb{Z}/_3\mathbb{Z}$, is the union of the encodings of P_1 and P_2.

4.2.1.6 Restriction

The encoding of P/x is the same as the encoding of P. That means the local definitions do not affect the encoding in the finite field modulo $p = 3, \mathbb{Z}/_3\mathbb{Z}$.

4.2.1.7 Clock relations

Let us recall the rewriting of the clock relations. Following the above encoding scheme, we can obtain the following encoding for derived operators on clocks. Here, z is a signal of type `event`:

$$\begin{array}{lll}
(z := \hat{x}) = (x = x) & \longrightarrow & z = x^2 \\
(x \hat{=} y) = (\hat{x} = \hat{y}) & \longrightarrow & x^2 = y^2 \\
z := (\texttt{ when } b) = (b \texttt{ when } b) & \longrightarrow & z = -b - b^2 \\
z := (x \hat{+} y) = (\hat{x} \texttt{ default } \hat{y}) & \longrightarrow & z = x^2 + (1-x^2)y^2 \\
z := (x \hat{*} y) = (\hat{x} \texttt{ when } \hat{y}) & \longrightarrow & z = x^2 y^2 \\
z := (x \hat{-} y) = (\texttt{ when } (\texttt{not } \hat{y} \texttt{ default } \hat{x})) & \longrightarrow & z = x^2(1-y^2)
\end{array}$$

The composition of the basic encodings of the SIGNAL primitive operators, presented in the above encoding scheme, constructs a polynomial dynamical system. Formally, the definition of PDS is given as follows:

Definition 12 (PDS) *A* PDS *is a system of equations which is organized into three subsys-*

tems of polynomial equations of the form:

$$\begin{cases} Q(X,Y) & = & 0 \\ X' & = & P(X,Y) \\ Q_0(X) & = & 0 \end{cases}$$

where:

- *X is a set of n variables, called state variables, represented by a vector in $(\mathbb{Z}/_3\mathbb{Z})^n$,*
- *Y is a set of m variables, called event variables, represented by a vector in $(\mathbb{Z}/_3\mathbb{Z})^m$,*
- *$X' = P(X,Y)$ is the evolution equation of the system. It can be considered as a vectorial function $[P_1,...,P_n]$ from $(\mathbb{Z}/_3\mathbb{Z})^{n+m}$ to $(\mathbb{Z}/_3\mathbb{Z})^n$,*
- *$Q(X,Y) = 0$ is the constraint equation of the system. It is a vectorial equation $[Q_1,...,Q_l]$,*
- *$Q_0(X) = 0$ is the initialization equation of the system. It is a vectorial equation $[Q_{0_1},...,Q_{0_n}]$.*

The equations of PDS have the following meaning:

- The initialization equation represents the initialization of signals in the program.
- The constraint equation is a set of equations which expresses the clock relations of the program. It is also considered as the invariant properties of the program.
- The evolution equation describes the evolution of state variables (corresponding to the *delay* operators) according to the logical time in the program. Therefore, it shows the dynamical behavior of the program.

4.2.1.8 Example

We consider the program `Altern` in Listing 4.1.

Listing 4.1 ALTERN in Signal

```
1 process ALTERN=
2 (? event A,B;
3  !)
4  (| X := not ZX
5   | ZX := X$1
```

```
6    | A ^= when X
7    | B ^= when ZX
8    |)
9  where boolean X, ZX init false
10 end;
```

It is translated in the PDS model with variables a, b, x and zx corresponding to the signals A, B, X and ZX, and a state variable $m.x$ introduced by the *delay* operator. The PDS model is given in Listing 4.2.

Listing 4.2 PDS of ALTERN

```
1 /* initial equation */
2 m.x = −1
3 /* evolution equation */
4 m.x' = x + (1 − x^2) ∗ m.x
5 /* constraint equation */
6 x = −zx;
7 zx = m.x ∗ x^2;
8 a^2 = −x − x^2;
9 b^2 = −zx − zx^2
```

4.3 Translation validation of PDSs

To apply the translation validation to the transformations on clocks of the SIGNAL compiler, we capture the clock semantics of the original program and its transformed counterpart by means of PDSs that is proposed by Marchand et al. [98] to build a formal verification framework of SIGNAL programs. The semantics of PDSs is considered as the common semantics framework in our translation validation. Then, we introduce a refinement relation which expresses the preservation of clock semantics, as a simulation relation between two PDSs. We provide a method to check the existence of that simulation based on model checking technique which is implemented as a library of SIGALI checker [52].

4.3.1 Definition of correct transformation: PDS refinement

Given a PDS model over the finite field $\mathbb{Z}/3\mathbb{Z}$, it can be viewed as an *intensional Labeled Transition System* (ILTS) [86] as in the following definition.

Definition 13 (ILTS) *An intensional labeled transition system is a structure* $L = (Q, Y, \mathscr{I}, \mathscr{T})$, *where* Q *is a set of states,* Y *is a set of m variables* $Y_1, ..., Y_m$, $\mathscr{I}$ *is a set of initial states, and*

$\mathscr{T} \subseteq Q \times \mathbb{Z}/_3\mathbb{Z}[Y] \times Q$ is the transition relation. Each transition is labeled by a polynomial over the set Y.

The ILTS representation of a PDS can be obtained directly from the set of state variables, event variables, initialization, constraint and evolution equations as follows:

- $Q = \mathscr{D}_X$, where $\mathscr{D}_X = \prod_{i\in[1,n]} \mathscr{D}_{x_i} = (\mathbb{Z}/_3\mathbb{Z})^n$ as the domain of a set of variables $X = (x_1,...,x_n)$
- $Y = Y, \mathscr{D}_Y = \prod_{i\in[1,m]} \mathscr{D}_{y_i} = (\mathbb{Z}/_3\mathbb{Z})^m$
- $\mathscr{I} = Sol(Q_0(X))$
- $(q, P_q(Y), q') \in \mathscr{T}$ where $P_q(Y) \equiv Q(q,Y) \oplus (P(q,Y) - q')$

We write $q \xrightarrow{P(Y)} q'$ (or for short $q \xrightarrow{P} q'$), instead of $(q, P(Y), q') \in \mathscr{T}$. Then ILTSs can be viewed as an "intensional" representation of classical LTSs, where the labels are tuples in $(\mathbb{Z}/_3\mathbb{Z})^m$: each arrow of the ILTS labeled by $P(Y)$ intensionally represents as many arrows labeled by some $y \in Sol(P(Y))$. The corresponding "extensional" LTS of an ILTS L is denoted by $Ext(L)$, meaning that each arrow of L labeled by $P(Y)$ is represented by the number (the number of solutions of $P(Y)$) of arrows labeled by $y \in Sol(P(Y))$.

For example, we consider the PDS model in Listing 4.2. Its ILTS representation is given as follows:

- $Q = \{-1, 0, +1\})$
- $Y = \{a, b, x, zx\}$
- $\mathscr{I} = Sol(m.x = -1) = \{0\}$
- $(q, P_q(Y), q') \in \mathscr{T}$ where $P_q(Y) \equiv Q(q,Y) \oplus (P(q,Y) - q')$,
  $Q(q,Y) = [x = -zx, zx = q * x^2, a^2 = -x - x^2, b^2 = -zx - zx^2]$, and
  $P(q,Y) = x + (1 - x^2) * q$

Definition 14 (Execution) *Let $L = (Q, Y, \mathscr{I}, \mathscr{T})$ be an* ILTS. *The infinite sequence $\sigma = q_0, y_0, q_1, y_1, q_2, y_2, ...$, where $q_i \in Q, y_i \in \mathscr{D}_Y$ for each $i \in \mathbb{N}$, is an execution of L if it satisfies the following requirements:*

- $q_0 \in \mathscr{I}$.

- *There exists a polynomial* $P(Y)$ *such that* $(q_i, P(Y), q_{i+1}) \in \mathscr{T} \wedge y_i \in Sol(P(Y))$ *for each* $i \in \mathbb{N}$.

We denote by $\sigma_{act} = y_0, y_1, y_2, \ldots.$ an action-based execution, $||L||, ||L||_{act}$ the sets of executions and action-based executions of the ILTS L, respectively.

Consider the two ILTSs $L_A = (Q_2, Y, \mathscr{I}_2, \mathscr{T}_2)$ and $L_C = (Q_1, Y, \mathscr{I}_1, \mathscr{T}_1)$, to which we refer respectively as the ILTS of the source program A and the ILTS of the compiled program C produced by the compiler. We assume that they have the same set of event variables. In case the set of event variables of the compiled model is different from the set of event variables of the source model, we consider only the set of common event variables and the different event variables are considered as *hiding events* as described in [115]. That means that we ensure that the clocks of common signals in the source and compiled programs are the same after the transformations.

In our case, the set of action-based executions models the clock semantics of the program. The set of action-based executions reflects the status of the clocks and clock relations in the program. The strongest notion of clock semantics preservation during compilation is that the source program A and its compiled program C have exactly the same clock semantics, meaning that their corresponding intensional labeled transition systems have the same set of action-based executions:

$$\forall \sigma_{act}.\ (\sigma_{act} \in ||L_C||_{act} \Leftrightarrow \sigma_{act} \in ||L_A||_{act}) \tag{4.1}$$

The compiled program C has fewer behaviors than the source program A. For instance, compilers do transformations, optimizations for removing or eliminating some behaviors of the source program (e.g., eliminating subexpressions, trivial clock constraints). To address these issues, we relax the requirement 4.1 as follows:

$$\forall \sigma_{act}.\ (\sigma_{act} \in ||L_C||_{act} \Rightarrow \sigma_{act} \in ||L_A||_{act}) \tag{4.2}$$

Requirement 4.2 says that all action-based executions of L_C are acceptable executions of L_A. And we say that L_C *refines* L_A w.r.t action-based executions, denoted by $L_C \sqsubseteq_{Pds} L_A$. In the next section we will present a method to establish the refinement between the two given ILTSs L_C and L_A.

4.3.2 Proving refinement by simulation

We now discuss an approach to automatically prove that a compiler preserves the clock semantics of the source program during its compilation, in the sense of refinement relation. Given two ILTSs L_A and L_C, we use a *symbolic simulation* [86] for the two ILTSs to establish that $L_C \sqsubseteq_{Pds} L_A$. The symbolic simulation satisfies the property that if there exists a symbolic simulation for (L_C, L_A) then $L_C \sqsubseteq_{Pds} L_A$.

Definition 15 (Symbolic simulation) *Let $L_C = (Q_1, Y, \mathscr{I}_1, \mathscr{T}_1)$ and $L_A = (Q_2, Y, \mathscr{I}_2, \mathscr{T}_2)$ be two* ILTS*s. A symbolic simulation for (L_C, L_A) is a binary relation $\mathscr{R} \subseteq Q_1 \times Q_2$ which satisfies the following properties:*

1. $\forall q_1 \in \mathscr{I}_1, \exists q_2 \in \mathscr{I}_2, (q_1, q_2) \in \mathscr{R}$.
2. *for any $(q_1, q_2) \in \mathscr{R}$ it holds that: if $q_1 \xrightarrow{P} q_1'$ there exists a finite set of transitions $(q_2 \xrightarrow{P_i} q_2^i)_{i \in I}$ (where I is a set of indexes) with*
   - $(P \Rightarrow \prod_{i \in I} P_i) \equiv 0$ *and*
   - $(q_1', q_2^i) \in \mathscr{R}, \forall i \in I$.

$(P \Rightarrow \prod_{i \in I} P_i) \equiv 0$ denotes that the polynomial $(P \Rightarrow \prod_{i \in I} P_i)$ is equivalent to the zero polynomial, which means that $Sol((P \Rightarrow \prod_{i \in I} P_i)) = Sol(0) = (\mathbb{Z}/_3\mathbb{Z})^m$ or $Sol(P) \subseteq Sol(\prod_{i \in I} P_i)$. Condition (1) asserts that every initial state of L_C is related to an initial state of L_A. According to condition (2), for every transition of the state q_1 which is labeled by the set of events (or actions) represented by $Sol(P(Y))$, there exist some transitions of the state q_2 which are labeled by the same set of events. And it states that every outgoing transition from q_1 must be matched by outgoing transitions from q_2. Thus, Definition 15 captures exactly classic action-based simulation definition of standard LTSs. Since symbolic simulation is closed under arbitrary unions, there is a greatest symbolic simulation. In the following parts, when we talk about symbolic simulation, we imply talking about the greatest symbolic simulation.

L_C is simulated by L_A (or equivalently, L_A simulates L_C), denoted by $L_C \preceq L_A$, if there exists a symbolic simulation for (L_C, L_A). Given two states $q_1 \in Q_1$ and $q_2 \in Q_2$, the state q_1 is simulated by q_2, denoted $q_1 \preceq q_2$, if there exists a symbolic simulation $\mathscr{R}$ for (L_C, L_A) with $(q_1, q_2) \in \mathscr{R}$. In that case, we say that the two states "q_1 and q_2 are similar".

Definition 16 *Let $L_C = (Q_1, Y, \mathscr{I}_1, \mathscr{T}_1)$ and $L_A = (Q_2, Y, \mathscr{I}_2, \mathscr{T}_2)$ be two* ILTS*s. We define a family of binary relations $\preceq_j \subseteq Q_1 \times Q_2$ by induction over $j \in \mathbb{N}$.*

- $\preceq_0 \triangleq Q_1 \times Q_2$.
- $q_1 \preceq_{(j+1)} q_2$ *iff* $\forall (q_1, P, q_1') \in \mathscr{T}_1$, *there exists a finite set of transitions* $(q_2, P_i, q_2^i)_{i\in I}$ *with* $(P \Rightarrow \Pi_{i\in I} P_i) \equiv 0 \wedge q_1' \preceq_j q_2^i, \forall i \in I$, *where* I *is a set of indexes.*

Based on the above definition, we can now have the following theorem which gives us a method to compute the greatest symbolic simulation for two ILTSs.

Theorem 2 *Let* $L_C = (Q_1, Y, \mathscr{I}_1, \mathscr{T}_1)$ *and* $L_A = (Q_2, Y, \mathscr{I}_2, \mathscr{T}_2)$ *be two* ILTSs.

1. *There exists a symbolic simulation for* (L_C, L_A) *iff there exists a simulation for the corresponding "extensional"* LTSs, $(Ext(L_C), Ext(L_A))$.
2. *Then for all* $q_1 \in Q_1$ *and* $q_2 \in Q_2$, $q_1 \preceq q_2$ *iff* $q_1 \left(\bigcap_{n\in\mathbb{N}} \preceq_n\right) q_2$, *where* $\left(\bigcap_{n\in\mathbb{N}} \preceq_n\right) = \preceq_0 \cap \preceq_1 \cap ... \cap \preceq_n$.

Proof 1 (1) *Following an approach which is similar to [86],*
(2) *Since the number of state variables, event variables and the value domain of a* PDS *are finite then its* ILTS *is finite. Symbolic simulation over a finite* ILTS *(therefore finitely branching) is the limit of nested projective equivalences. Thus we can use the same proof method as in [100] for strong simulation. We omit the proof here.*

The use of a symbolic simulation as a proof method to establish the refinement between the two given models L_C and L_A is stated in the following theorem.

Theorem 3 *Let* $L_C = (Q_1, Y, \mathscr{I}_1, \mathscr{T}_1)$ *and* $L_A = (Q_2, Y, \mathscr{I}_2, \mathscr{T}_2)$ *be two* ILTSs. *If there exists a symbolic simulation for* (L_C, L_A), *then* $L_C \sqsubseteq_{Pds} L_A$.

Proof 2 *The proof of Theorem 3 is trivial, following Lemma 1.*

Lemma 1 *Let* L_C *and* L_A *be* ILTSs, $\mathscr{R}$ *is a symbolic simulation for* (L_C, L_A), *and* $(q_1, q_2) \in \mathscr{R}$. *Then for each infinite (or finite) execution* $\sigma_1 = q_{0,1}, y_{0,1}, q_{1,1}, y_{1,1}, q_{2,1}, y_{2,1}, ...$ *starting in* $q_{0,1} = q_1$ *there exists an execution* $\sigma_2 = q_{0,2}, y_{0,2}, q_{1,2}, y_{1,2}, q_{2,1}, y_{2,2}, ...$ *from state* $q_{0,2} = q_2$ *of the same length such that* $(q_{j,1}, q_{j,2}) \in \mathscr{R}$ *and* $y_{j,1} = y_{j,2}$ *for all* j.

Proof 3 *Let* $\sigma_1 = q_{0,1}, q_{1,1}, q_{2,1}, ...$ *be an execution in* L_C *starting in* $q_1 = q_{0,1}$ *and assume* $(q_1, q_2) \in \mathscr{R}$. *We can define a corresponding execution in* L_A *starting in* $q_2 = q_{0,2}$ *with the same length (in case the execution* σ_1 *is finite), where the transitions* $q_{i,1} \longrightarrow q_{i+1,1}$ *are matched by transitions* $q_{i,2} \longrightarrow q_{i+1,2}$ *such that* $(q_{i+1,1}, q_{i+1,2}) \in \mathscr{R}$. *We use the induction method on* i *to prove it.*

- *Base case: $i = 0$. It follows directly from $(q_1, q_2) \in \mathscr{R}$ in case q_1 is a terminal state. If there is a transition $q_{0,1} \xrightarrow{P(Y)} q_{1,1}$ such that $y_{0,1} \in Sol(P(Y))$ then there exists a finite set of transitions $(q_{0,2} \xrightarrow{P_j} q^j_{1,2})_{j\in J}$ with $Sol(P) \subseteq Sol(\prod_{j\in J} P_j)$ and $(q_{1,1}, q^j_{1,2}) \in \mathscr{R}, \forall j \in J$. Because $Sol(P) \subseteq Sol(\prod_{j\in J} P_j)$, there exists a polynomial $P_j(Y)$ such that $y_{0,1} \in Sol(P_j)$, and the transition $q_{0,1} \xrightarrow{y_{0,1}} q_{1,1}$ can be matched by the transition $q_{0,2} \xrightarrow{y_{0,1}} q^j_{1,2}$ with $(q_{1,1}, q^j_{1,2}) \in \mathscr{R}$. This yields the execution fragment $q_{0,2}, y_{0,2}, q_{1,2}$ with $y_{0,1} = y_{0,2}$ in L_A.*

- *Induction step: Assume $i > 0$ and that the execution $q_2, y_{0,2}, q_{1,2}, y_{1,2}, q_{2,2}, y_{2,2}, \ldots, q_{i,2}$ is already constructed with $(q_{k,1}, q_{k,2}) \in \mathscr{R}$ and $y_{k,1} = y_{k,2}$ for $k = 0, \ldots, i$. If σ_1 has length i and $q_{i,1}$ is a terminal state, then the execution fragment $\sigma_2 = q_2, y_{0,2}, q_{1,2}, y_{1,2}, q_{2,2}, y_{2,2}, \ldots, q_{i,2}$ is an execution fragment with the same length which is state-wise related to σ_1. Now we assume that $s_{i,1}$ is not terminal. We consider the step $q_{i,1} \xrightarrow{P(Y)} q_{i+1,1}$ with $y_{i,1} \in Sol(P(Y))$ in σ_1. Since $(q_{i,1}, q_{i,2}) \in \mathscr{R}$, there exists a finite set of transitions $(q_{i,2} \xrightarrow{P_j} q^j_{i+1,2})_{j\in J}$ with $Sol(P) \subseteq Sol(\prod_{j\in J} P_j)$ and $(q_{i+1,1}, q^j_{i+1,2}) \in \mathscr{R}, \forall j \in J$. Because $Sol(P) \subseteq Sol(\prod_{j\in J} P_j)$, there exists a polynomial $P_j(Y)$ such that $y_{i,1} \in Sol(P_j)$, and the transition $q_{i,1} \xrightarrow{y_{i,1}} q_{i+1,1}$ can be matched by the transition $q_{i,2} \xrightarrow{y_{i,1}} q^j_{i+1,2}$ with $(q_{i+1,1}, q^j_{i+1,2}) \in \mathscr{R}$. This yields the execution fragment $q_2, y_{0,2}, q_{1,2}, y_{1,2}, \ldots, q_{i,2}, y_{i,2}, q_{i+1,2}$ which is state-wise related to the execution σ_1 and $y_{i,1} = y_{i,2}$ in L_A.*

4.3.3 Composition of compilation phases

Let Cp be an unverified compiler, the following process defines a derived compiler such that it is formally verified.

```
1 if (Cp(A) == Error) return Error
2 else
3 {
4    if (L_IR(A) ⊑_Pds L_A) return IR(A)
5    else return Error
6 }
```

where $Cp(A)$ is the compilation of A to either compiled code (written as $Cp(A) = IR(A)$) or compilation errors (written as $Cp(A) = \texttt{Error}$), L_A and $L_{IR(A)}$ are the intensional labeled transition systems of A and $IR(A)$, respectively.

Compilation is always decomposed into several phases of transformations and optimizations through intermediate representations. Thus it is better to decompose the verification process too. Fortunately, our verification process can be decomposed well thanks to the transitive property of symbolic simulation. Let L_A, L_I and L_C be three ILTSs, if $L_I \preceq L_A$ and $L_C \preceq L_I$ then $L_C \preceq L_A$ (the proof is trivial based on the definition of symbolic simulation). We assume that there are two compilation phases $Cp1$ and $Cp2$ from the source program A to the compiled program I and from the compiled program I to other compiled program C, respectively. Consider the composition compilation as follows:

```
1  if (Cp1(A) == Error) return Error
2  else
3  {
4     if (L_I ⊑_Pds L_A)
5     {
6        if (Cp2(I) == Error) return Error
7        else
8        {
9           if (L_C ⊑_Pds L_I) return C
10          else return Error
11       }
12    }
13    else return Error
14 }
```

It is obvious to see that the compilation process from A to C is formally verified.

4.3.4 Implementation with SIGALI

In this section, we discuss how to implement the proof method with symbolic simulation using the companion model-checker of the POLYCHRONY toolset, SIGALI.

4.3.4.1 Implementation

We design a validator that is integrated into the *clock calculation and Boolean abstraction* phase of the SIGNAL compiler. The validator which is depicted in Figure 4.2 works as follows. It takes the input program and its transformed counterpart to construct the corresponding PDSs. These PDSs are given as the input of the model-checker SIGALI. In the checking phase, SIGALI will answer whether there exists a symbolic simulation for two PDSs.

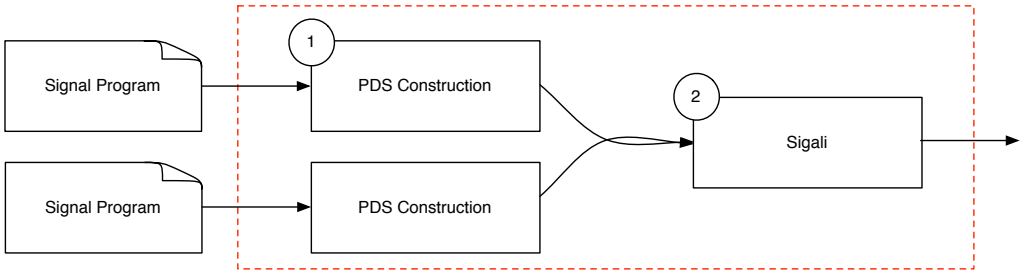

Fig. 4.2 The PDS translation validation

Symbolic simulation can be implemented as an extended library of SIGALI. We represent an ILTS in the more specific form $L = (X, X', Y, \mathscr{I}, \mathscr{T})$, where:

- X, X' and Y are the sets of state and event variables as in the PDS,
- $\mathscr{I}(X) = Q_0(X)$ is the polynomial representing the set of initial states, $Sol(I)$,
- $\mathscr{T}(X,Y,X') \equiv Q(X,Y) \oplus (P(X,Y) - X')$ is the polynomial representing the set of transitions.

In SIGALI, polynomials are internally represented as *Ternary Decision Diagrams* (TDD) [52] which are an extension of *Binary Decision Diagrams* (BDD) [32]. They are convenient for an efficient manipulation of the polynomial equation systems. Theorem 2 gives us an iterative algorithm to compute the greatest symbolic simulation for two intensional labeled transition systems. It can be obtained by computing the convergence of the sequence $(\mathscr{R}_j)_{j\in\mathbb{N}}$ as in Listing 4.3 which can be efficiently implemented with the fixed-point computation of the SIGALI kernel. The correctness of Listing 4.3 is proved by the following proposition.

Listing 4.3 Compute symbolic simulation

```
1  Input: L_C = (X_1, X_1', Y, I_1, T_1), L_A = (X_2, X_2', Y, I_2, T_2)
2  Output: R(X_1, X_2)
3  R_0(X_1, X_2) ≡ 0
4  while (R_j(X_1, X_2) is not convergent) do
5  {
6      R_{j+1}(X_1, X_2) is the canonical generator of the ≡-class of:
7      R_j(X_1, X_2) ⊕
8      ∀X_1' ∀Y [(T_1(X_1, Y, X_1') ⇒ ∃X_2' (T_2(X_2, Y, X_2') ⊕ R_j(X_1', X_2')))]
9  }
10 if (∀X_1 [(I_1(X_1) ⇒ ∃X_2 (I_2(X_2) ⊕ R(X_1, X_2)))])
```

```
    return ℛ(X1,X2)
else
    return ℛ(X1,X2) ≡ 1
```

Proposition 1 *For all* $j \in \mathbb{N}, \mathscr{R}_j(x_1,x_2) = 0$ *iff* $x_1 \preceq_j x_2$.

Proof 4 $\Rightarrow$) *We use an induction proving method over* j. *It holds obviously with* $j = 0$. *Assume that we have* $\mathscr{R}_{j+1}(x_1,x_2) = 0$ *and let* $x_1 \xrightarrow{P} x_1'$ *be a transition in* L_C. *It is clear that* $P(Y) \equiv \mathscr{T}_1(x_1,Y,x_1')$. *We define the polynomial* $Q(Y) \equiv \exists x_2' \mathscr{T}_2(x_2,Y,x_2') \oplus \mathscr{R}_j(x_1',x_2')$, $\mathscr{R}_j$ *being computed in Listing 4.3 above. This polynomial captures the set* $\{y | \exists x_2 \xrightarrow{P_i} x_2^i, P_i(y) = 0 \wedge x_1' \preceq_j x_2^i\}$. *By the definition of* $\mathscr{R}_{j+1}$, *the* y *value is in* $Sol(\mathscr{T}_1(x_1,Y,x_1'))$. *We have* $Sol(P(Y)) \subseteq \bigcup_i Sol(P_i)$, *which means* $x_1 \preceq_{(j+1)} x_2$.
$\Leftarrow$) *We can apply again an induction method over* j *similar to the proof of Theorem 2. Thus, we omit it here.*

Proposition 2 *Algorithm in Listing 4.3 terminates and at the end,* $\mathscr{R}(x_1,x_2) = 0$ *if and only if* $x_1 \preceq x_2$.

Proof 5 *Termination is guaranteed by the fact that relations* $\mathscr{R}_j$ *are finite and nested. The second statement is a corollary of Proposition 1 and Theorem 2.*

Listing 4.4 Symbolic simulation implementation in SIGALI

```
% internal function P1 => P2 %
def implies(P1,P2):
    union(complementary(P1),P2);

def states_simulation(X_1,Y_1,X_1_nexts,Rel_1,
X_2,Y_2,X_2_nexts,Rel_2) :
    with
       % define utility variables %
       Y_1_bar = diff_lvar(Y_1,Y_2),
       Y_2_bar = diff_lvar(Y_2,Y_1),
       Y = diff_lvar(Y_1,Y_1_bar),
       X_1_X_2 = union_lvar(X_1,X_2),
       X_1_nexts_Y_1_bar = union_lvar(X_1_nexts,Y_1_bar),
       X_2_nexts_Y_2_bar = union_lvar(X_2_nexts,Y_2_bar),
       X_1_X_2_nexts = union_lvar(X_1_nexts,X_2_nexts)
    do
    % compute the simulation %
```

```
18    loop x =
19     intersection(x,
20          forall(forall(exist(implies(Rel_1,
21           exist(intersection(Rel_2,
22            rename(x,X_1_X_2,X_1_X_2_nexts)),
23             X_2_nexts_Y_2_bar)),Y_1_bar),Y),X_1_nexts))
24     init 0;
25
26 def ilts_simulation(S_I1,S_I2) :
27    with
28       % get the components of iLTS1 %
29       I_1 = initial_I(S_I1),
30       X_1 = state_var_I(S_I1),
31       X_1_nexts = state_var_next_I(S_I1),
32       Y_1 = event_var_I(S_I1),
33       Rel_1 = trans_rel_I(S_I1),
34       % get the components of iLTS2 %
35       I_2 = initial_I(S_I2),
36       X_2_d = state_var_I(S_I2),
37       X_2_nexts_d = state_var_next_I(S_I2),
38       Y_2 = event_var_I(S_I2),
39       Rel_2_d = trans_rel_I(S_I2),
40       % rename the states variables %
41       X_2 = declare_suff(X_2_d),
42       X_2_nexts = declare_suff(X_2_nexts_d),
43       Rel_2 = rename(Rel_2_d,union_lvar(X_2_d,X_2_nexts_d),
44                      union_lvar(X_2,X_2_nexts)),
45       states_sim = states_simulation(X_1,Y_1,X_1_nexts,
46                     Rel_1,X_2,Y_2,X_2_nexts,Rel_2)
47    do
48    % compute the systems simulation %
49    intersection(states_sim,
50                 forall(implies(I_1,exist(intersection(
51                 states_sim,I_2),X_2)),X_1));
```

We provide an implementation of algorithm in Listing 4.3 to compute the greatest simulation for two iLTSs. The implementation uses some useful functions from the kernel of the checker SIGALI. The inputs are the concrete and abstract iLTSs *S_I1* and *S_I2*, respectively.

4.3.4.2 Experimental results

Table 4.1 shows some experimental results verifying the transformations of the SIGNAL compiler with a simulation based proof method. The experimental results deal with the complexity of the symbolic simulation computation. All the examples are available in the online examples of the POLYCHRONY toolset. In the X, Y, and "Correct" columns, we write the numbers of state variables, event variables and the correctness of the compiler transformations, respectively (hence, the transition relation $\mathscr{T}(X,Y,X')$ will have $2X+Y$ variables). We measure the complexity of the symbolic simulation by the size of fix point computation in Listing 4.3 (in terms of the number of TDD nodes that we need to represent the manipulation of polynomial equation systems). The number of TDD nodes is shown in SIGALI model checker only when it is big enough, so for the tests whose numbers of TDD nodes are not shown we write "Small" to indicate that the computation effort is light. We denote $\mathscr{R}_1(X_1,X_2), \mathscr{R}_2(X_1,X_2), \mathscr{R}_3(X_1,X_2)$ symbolic simulations for (`A_TRA.z3z`,`A.z3z`), (`A_BOOL_TRA.z3z`,`A_TRA.z3z`) and (`A_SEQ_TRA.z3z`, `A_BOOL_TRA.z3z`), respectively, for the compilation of program A.

4.4 Translation validation of clock models

We provide another approach to prove the preservation of the clock semantics for every transformation of the compiler based on the translation validation approach. We use clock model as the common semantic framework to represent the clock semantics of source program and its compiled program. A refinement relation is formally defined to expresses the preservation of clock semantics, as a relation on clock models. We write $\Phi(C) \sqsubseteq_{clk} \Phi(A)$ to denote that $\Phi(C)$ is a refinement of $\Phi(A)$.

4.4.1 Clock model of SIGNAL program

In this section, we describe the clock semantics of a program in terms of a first-order logic formula. Let us consider the semantics of the *sampling* operator $y := x$ `when` b. At any logical instant, the signal y holds the value of x if the following conditions are satisfied: x holds a value, and b is present and holds the value `true`; otherwise, it holds no value. Hence, to represent the underlying control conditions, we need to represent the statuses: *present* with the value `true`, *present* with the value `false` and *absent* for the signal b, and the statuses: *present* and *absent* for the signal x. This section explores a method to construct the control model of a program as an abstraction of the clock semantics, called clock model,

Name	X	Y	$\mathscr{R}_1(X_1,X_2)$ TDD nodes	$\mathscr{R}_2(X_1,X_2)$ TDD nodes	$\mathscr{R}_3(X_1,X_2)$ TDD nodes	Correct
MOUSE.z3z	2	5	Small	Small	Small	Yes
_TRA.z3z	2	5				
_BOOL_TRA.z3z	2	6				
_SEQ_TRA.z3z	2	6				
RAILROADCROSSING.z3z	2	40	Small	Small	Small	Yes
_TRA.z3z	2	40				
_BOOL_TRA.z3z	2	39				
_SEQ_TRA.z3z	2	39				
CHRONOMETER.z3z	6	33	Small	Small	Small	Yes
_TRA.z3z	6	33				
_BOOL_TRA.z3z	6	37				
_SEQ_TRA.z3z	6	37				
ALARM.z3z	19	45	3775163	3810301	4721454	Yes
_TRA.z3z	19	45				
_BOOL_TRA.z3z	19	53				
_SEQ_TRA.z3z	19	53				

Table 4.1 Translation validation of PDSs: Experimental results

which is the computational model of our translation validation approach.

In SIGNAL, clocks play a much more important role than in other synchronous languages, they are used to express the underlying control (i.e., the synchronization between signals) for any conditional definition. This differs from LUSTRE, where all clocks are built by sampling the fastest clock. For instance, we consider again the basic process corresponding to the primitive operator *sampling*, $y := x$ `when` b, where x and y are numerical signals, and b is a Boolean signal. To express the control, we need to represent the status of the signals x, y and b at a given instant. In this example, we use a Boolean variable $\hat{x}$ to capture the status of x:

- $(\hat{x} = \texttt{true})$ means x is present.
- $(\hat{x} = \texttt{false})$ means x is absent.

In the same way, the Boolean variable $\hat{y}$ captures the status of y. For the Boolean signal b, two Boolean variables $\hat{b}$ and $\tilde{b}$ are used to represent its status:

- $(\hat{b} = \texttt{true} \wedge \tilde{b} = \texttt{true})$ means b is present and holds a value `true`.
- $(\hat{b} = \texttt{true} \wedge \tilde{b} = \texttt{false})$ means b is present and holds a value `false`.
- $(\hat{b} = \texttt{false})$ means b is absent.

Hence, at a given instant, the implicit control relations of the basic process above can be encoded by the following formula:

$$\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \tilde{b})$$

Let $X = \{x_1, ..., x_n\}$ be the set of all signals in program P consisting of input, output, register (corresponding to *delay* operator), and local signals, denoted by I, O, R and L, respectively. With each signal x_i, based on the encoding scheme proposed in [57, 61, 63], we attach a Boolean variable $\widehat{x_i}$ to encode its clock and a variable $\widetilde{x_i}$ of same type as x_i to encode its value. If $x_i \in R$ then we introduce a memorization variable to store the initialized, previous and next values. Formally, the abstract values which represent the clock semantics of the program can be computed using the following functions:

$\hat{}: X \longrightarrow \mathbb{B}$ associates a signal with a Boolean value,

$\tilde{}: X \longrightarrow \mathbb{D}$ associates a signal with a value of same type as the signal.

The composition of SIGNAL processes corresponds to logical conjunction. Thus the clock model of P will be a conjunction $\Phi(P) = \bigwedge_{i=1}^{n} \phi(eq_i)$ whose atoms are $\widehat{x_i}, \widetilde{x_i}$, where $\phi(eq_i)$ is the abstraction of statement eq_i (using the SIGNAL primitive operators), and n is the number of statements in the program. In the following, we present the abstraction corresponding to each SIGNAL operator.

4.4.1.1 Stepwise functions

The functions which apply on signal values in the primitive *stepwise functions* are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`), and numerical operators (`+`, `-`, `*`, `/`). In our experience working with the SIGNAL compiler, it performs very few arithmetical optimizations and leaves most of the arithmetical expressions intact. Every variable is determinable by the inputs, memorizable values, otherwise program cannot be compiled. This suggests that most of the implications will hold independently of the features of the numerical comparison functions and numerical operators and we can replace the operations by *uninterpreted functions*. Following the encoding procedure of [3], for every numerical comparison function and numerical operator (denoted by $\square$) occurring in an equation, we perform the following rewriting:

- Replace each $x \square y$ by a new variable $v^i_\square$ of a type equal to that of the value returned by $\square$. Two stepwise functions $x \square y$ and $x' \square y'$ are replaced by the same variable $v^i_\square$ iff x, y are identical to x' and y', respectively.
- For every pair of newly added variables $v^i_\square$ and $v^j_\square$, $i \neq j$, corresponding to the non-identical occurrences $x \square y$ and $x' \square y'$, add the implication $(\tilde{x} = \widetilde{x'} \wedge \tilde{y} = \widetilde{y'}) \Rightarrow \widetilde{v^i_\square} = \widetilde{v^j_\square}$ into the abstraction $\Phi(P)$.

The abstraction $\phi(y := f(x_1, ..., x_n))$ of stepwise functions is defined by induction as follows:

- $\phi(\texttt{true}) = \texttt{true}$ and $\phi(\texttt{false}) = \texttt{false}$.
- $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \tilde{x}))$ if x and y are Boolean signals. $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \tilde{x})) \wedge (\hat{x} \Rightarrow \tilde{x})$ if x is an `event` signal.
- $\phi(y := x_1 \text{ and } x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \widetilde{x_1} \wedge \widetilde{x_2}))$.
- $\phi(y := x_1 \text{ or } x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \widetilde{x_1} \vee \widetilde{x_2}))$.
- $\phi(y := x_1 \square x_2) = (\hat{y} \Leftrightarrow \widehat{v^i_\square} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\tilde{y} = \widetilde{v^i_\square}))$.

4.4.1.2 Delay

Considering the *delay* operator, $y := x\$1 \texttt{ init } a$, its encoding $\phi(y := x\$1 \texttt{ init } a)$ contributes to $\Phi(P)$ with the following conjunct:

$$\begin{array}{ll} & (\hat{y} \Leftrightarrow \hat{x}) \\ \wedge & (\hat{y} \Rightarrow ((\tilde{y} = m.x) \wedge (m.x' = \tilde{x}))) \\ \wedge & (m.x_0 = a) \end{array}$$

This encoding requires that at any instant, signals x and y have the same status (present or absent). To encode the value of the output signal as well, we introduce a memorization variable $m.x$ that stores the last value of x. The next value of $m.x$ is $m.x'$ and it is initialized to a in $m.x_0$.

4.4.1.3 Merge

The encoding of the *merge* operator, $y := x \texttt{ default } z$, contributes to $\Phi(P)$ with the following conjunct:

$$\begin{array}{ll} & (\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})) \\ \wedge & (\hat{y} \Rightarrow ((\hat{x} \wedge \tilde{y} = \tilde{x}) \vee (\neg\hat{x} \wedge \tilde{y} = \tilde{z}))) \end{array}$$

4.4.1.4 Sampling

The encoding of the *sampling* operator, $y := x \texttt{ when } b$, contributes to $\Phi(P)$ with the following conjunct:

$$\begin{array}{ll} & (\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \tilde{b})) \\ \wedge & (\hat{y} \Rightarrow (\tilde{y} = \tilde{x})) \end{array}$$

4.4.1.5 Composition

Consider the composition of two processes P_1 and P_2. Its abstraction $\phi(P_1|P_2)$ is defined as follows:

$$\phi(P_1) \wedge \phi(P_2)$$

4.4.1.6 Restriction

Restriction process P_1/x restricts the scope of the signal x to the process P. In our abstraction the scope of signal declarations is not considered. The abstraction is given as follows:

$$\Phi(P_1/x) = \Phi(P_1)$$

4.4.1.7 Clock relations

Given the above rules, we can obtain the following abstraction for derived operators on clocks. Here, z is a signal of type *event*:

- $\phi(z := \hat{x}) = (\hat{z} \Leftrightarrow \hat{x}) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(x \hat{=} y) = \hat{x} \Leftrightarrow \hat{y}$
- $\phi(z := x \hat{+} y) = (\hat{z} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(z := x \hat{*} y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(z := x \hat{-} y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(z := \texttt{when}\ b) = (\hat{z} \Leftrightarrow (\hat{b} \wedge \tilde{b})) \wedge (\hat{z} \Rightarrow \tilde{z})$

4.4.1.8 Example

Applying the abstraction rules above, the clock semantics of the SIGNAL program DEC in Listing 3.3 is represented by the following first-order logic formula $\Phi(\texttt{DEC})$, where $ZN <= 1$ and $ZN - 1$ are replaced by two fresh variables $ZN1$ and $ZN2$, respectively. Two uninterpreted function symbols $v^1_{<=}$ and v^1_{-} are used to encode the stepwise functions $ZN1 := ZN <= 1$ and $ZN2 := ZN - 1$.

$$\begin{aligned}
& (\widehat{FB} \Leftrightarrow \widehat{ZN1} \wedge \widetilde{ZN1}) \\
\wedge\ & (\widehat{ZN1} \Leftrightarrow \widehat{v^1_{<=}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN1} \Rightarrow (\widetilde{ZN1} = \widetilde{v^1_{<=}})) \\
\wedge\ & (\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \tilde{N})) \wedge (m.N_0 = 1) \\
\wedge\ & (\widehat{N} \Leftrightarrow \widehat{FB} \vee \widehat{ZN2}) \wedge (\widehat{N} \Rightarrow ((\widehat{FB} \wedge \tilde{N} = \widetilde{FB}) \vee (\neg\widehat{FB} \wedge \tilde{N} = \widetilde{ZN2}))) \\
\wedge\ & (\widehat{ZN2} \Leftrightarrow \widehat{v^1_{-}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN2} \Rightarrow (\widetilde{ZN2} = \widetilde{v^1_{-}}))
\end{aligned}$$

In the following sections, we denote input, output, register, memorization and local variables in a clock model by $I_{clk}, O_{clk}, R_{clk}, M_{clk}$ and L_{clk}, respectively. Note that the memorization

variables are introduced only by the translation into clock models, they are not original in the SIGNAL programs.

4.4.1.9 Clock configuration

Consider a clock model $\Phi(P)$ of the program P. Let $\hat{X} = I_{clk} \cup O_{clk} \cup R_{clk} \cup M_{clk} \cup L_{clk}$ be the set of variables which is used to construct the clock model $\Phi(P)$. A *clock configuration* $\widehat{I}$ is an interpretation over $\hat{X}$ such that it is a model of the first-order logic formula $\Phi(P)$. For instance,

$$(\widehat{FB} \mapsto \texttt{true}, \widehat{N} \mapsto \texttt{true}, \widehat{ZN} \mapsto \texttt{true}, \widetilde{FB} \mapsto 6, \tilde{N} \mapsto 6, \widetilde{ZN} \mapsto 1)$$

is a clock configuration of $\Phi(\texttt{DEC})$. But the following interpretation

$$(\widehat{FB} \mapsto \texttt{true}, \widehat{N} \mapsto \texttt{true}, \widehat{ZN} \mapsto \texttt{false}, \widetilde{FB} \mapsto 6, \tilde{N} \mapsto 6, \widetilde{ZN} \mapsto 1)$$

is not a clock configuration since it is not a model of the clock model. The set of all clock configurations of $\Phi(P)$, denoted by $\mathscr{E}_{\Phi(P)}$, is given as follows:

$$\mathscr{E}_{\Phi(P)} = \{\widehat{I} \mid \widehat{I} \models \Phi(P)\}$$

4.4.2 Soundness of clock model

We will show that the clock model is sound in terms of preservation of the clock semantics of the abstracted program. That means if the clock model satisfies a property defined over the clocks, then the abstracted SIGNAL program also satisfies this property.

4.4.2.1 Clock semantics of SIGNAL program

We rely on the basic elements of *trace semantics* [70] to define the clock semantics of a synchronous data-flow program and the soundness proof method in [61]. Let $X = \{x_1, x_2, ..., x_n\}$ be a finite set of typed variables. For each $x_i \in X$, we use $\mathbb{D}_{x_i}$ to denote its domain of values, and $\mathbb{D}_{x_i}^{\perp} = \mathbb{D}_{x_i} \cup \{\perp\}$ to denote its domain of values with absent value, where $\perp \notin \mathbb{D}_{x_i}$ denotes the absent value. Then, the domain of values of X with absent value is defined as follows:

$$\mathbb{D}_X^{\perp} = \bigcup_{i=1}^{n} \mathbb{D}_{x_i} \cup \{\perp\}$$

Definition 17 (Clock events) *Given a non-empty set X, the set of clock events on X, denoted by $\mathscr{E}c_X$, is the set of all possible interpretations I over X.*

An interpretation I is an assignment of values from X to $\mathbb{D}_X^{\perp}$. The assignment $I(x) = \perp$ means x holds no value while $I(x) = v$ means that x holds the value v, and $I(X)$ is defined as follows:

$$I(X) = (I(x_1), I(x_2), ..., I(x_n)), x_i \in X$$

For example, consider a program whose signals are $X = \{x, b\}$ where x is an integer signal and b is Boolean signal, the set of clock events is given as follows, where v is an integer value:

$$\begin{aligned} \mathscr{E}c_X = \quad & \{(x \mapsto \perp, b \mapsto \perp), (x \mapsto \perp, b \mapsto \texttt{false}), \\ & (x \mapsto \perp, b \mapsto \texttt{true}), (x \mapsto v, b \mapsto \texttt{false}), \\ & (x \mapsto v, b \mapsto \texttt{true}), (x \mapsto v, b \mapsto \perp)\} \end{aligned}$$

Then at a given instant, the signals clock information is one of these clock events. By convention, the set of clock events of the empty set is defined as the empty set $\mathscr{E}c_{\emptyset} = \emptyset$.

Definition 18 (Clock traces) *Given a non-empty set X, the set of clock traces on X, denoted by $\mathscr{T}c_X$, is defined by the set of functions T_c defined from the set $\mathbb{N}$ of natural numbers to $\mathscr{E}c_X$, denoted by $T_c : \mathbb{N} \longrightarrow \mathscr{E}c_X$.*

The natural numbers represent the instants, $t = 0, 1, 2, ...$, a trace T_c is a chain of clock events. We denote the interpreted value of a variable x_i at instant t by $T_c(t)(x_i)$. Considering the above example:

$$(0, (x \mapsto \perp, b \mapsto \perp)), (1, (x \mapsto 1, b \mapsto \texttt{false})), ...$$

is one of the possible clock traces on $X = \{x, b\}$, and $T_c(0)(x) = T_c(0)(b) = \perp$

Definition 19 (Restriction clock trace) *Given a non-empty set X, a subset $X_1 \subseteq X$, and a clock trace T_c being defined on X, the restriction of T_c onto X_1 is denoted by $X_1.T_c$. It is defined as $X_1.T_c : \mathbb{N} \longrightarrow \mathscr{E}c_{X_1}$ such that $\forall t \in \mathbb{N}, \forall x \in X_1, X_1.T_c(t)(x) = T_c(t)(x)$.*

Let X be the set of all signals in program P. We write $[[P]]_c$ to denote the clock semantics of P which is defined as the set of all possible clock traces on X. For any subset $X_1 \subseteq X$, the set of all restriction clock traces on X_1 defines the clock semantics of P on X_1, denoted by $([[P]]_c)_{\backslash X_1}$. Table 4.2 shows the clock semantics of the SIGNAL processes corresponding to the primitive operators. In the clock semantics of the *delay* operator, inf is the infimum (or the *greatest lower bound*) of the subset of instants such that x is present, sup is the supremum

Process P	Clock semantics $[[P]]_c$
$y := f(x_1,...,x_n)$	$\{T_c \in \mathscr{T}c_{\{y,x_1,...,x_n\}} \mid \forall t \in \mathbb{N}, (\forall i, T_c(t)(x_i) = T_c(t)(y) = \bot)$ or $T_c(t)(y) = v_y$ and $T_c(t)(x_i) = v_{x_i}$ and $v_y = f(v_{x_1},...,v_{x_n})\}$
$y := x\$1$ `init` a	$\{T_c \in \mathscr{T}c_{\{x,y\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = T_c(t)(y) = \bot)$ or $(T_c(t)(x) \neq \bot$ and $T_c(t)(y) \neq \bot$ and $T_c(t_0)(y) = a$ and $\forall t \geq t_0$ $T_c(t)(y) = T_c(t_-)(x))$ with $t_0 = inf\{t' \mid T_c(t')(x) \neq \bot\}$, $t_- = sup\{t' \mid t' < t \wedge T_c(t')(x) \neq \bot\}\}$
$y := x$ `when` b	$\{T_c \in \mathscr{T}c_{\{x,y,b\}} \mid \forall t \in \mathbb{N}, (T_c(t)(b) =$ `true` and $T_c(t)(y) = T_c(t)(x) = \bot)$ or $(T_c(t)(b) =$ `true` and $T_c(t)(y) = T_c(t)(x) = v)$ or $(T_c(t)(b) =$ `false` and $T_c(t)(y) = T_c(t)(x) = \bot)$ or $(T_c(t)(b) =$ `false` and $T_c(t)(x) = v$ and $T_c(t)(y) = \bot)$ or $(T_c(t)(b) = \bot$ and $T_c(t)(x) = T_c(t)(y) = \bot)$ or $(T_c(t)(b) = \bot$ and $T_c(t)(x) = v$ and $T_c(t)(y) = \bot)\}$
$y := x$ `default` z	$\{T_c \in \mathscr{T}c_{\{x,y,z\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = v$ and $T_c(t)(y) = v)$ or $(T_c(t)(x) = \bot$ and $T_c(t)(z) = v$ and $T_c(t)(y) = v)$ or $(T_c(t)(x) = T_c(t)(z) = T_c(t)(y) = \bot)\}$
$P_1 \mid P_2$	$\{T_c \in \mathscr{T}c_{X_1 \cup X_2} \mid X_1.T_c \in [[P_1]]_c$ and $X_2.T_c \in [[P_2]]_c\}$ where $[[P_1]]_c \subseteq \mathscr{T}c_{X_1}, [[P_2]]_c \subseteq \mathscr{T}c_{X_2}$
P_1/x	$\{T_c \in \mathscr{T}c_{X_1 \setminus \{x\}} \mid \exists T_{c1} \in [[P_1]]_c, (X_1 \setminus \{x\}).T_{c1} = T_c\}$ where $[[P_1]]_c \subseteq \mathscr{T}c_{X_1}$

Table 4.2 Clock semantics of the SIGNAL primitive operators

(or *least upper bound*) of the subset of instants such that they are smaller than the instant t and x is present. For instance, the clock semantics of the basic process corresponding to *sampling* operator is the following set of clock traces, where x, y are numerical signals and b is Boolean signal:

$T_c = \{(0,(x_0,b_0,y)),...,(i,(x_i,b_i,y_i)),...\}$ such that

$\forall i, (x_i,b_i,y_i) \in \{(\bot,\bot,\bot),(\bot,$`false`$,\bot),(\bot,$`true`$,\bot),(v,\bot,\bot),(v,$`false`$,\bot),(v,$`true`$,v)\}$

4.4.2.2 Concrete clock semantics

Let $\Phi(P)$ be the clock model of the program P. We now define the *concrete clock semantics* of a clock model based on the notion of clock configurations. Given a clock configuration $\hat{I}$, the set of clock events according to $\hat{I}$ is the set of interpretations I such that for every signal x_i, if x_i holds a value then $\widehat{x_i}$ has the value `true` (meaning x_i is present), and $\widetilde{x_i}$ holds the same value as x_i. Otherwise, $\widehat{x_i}$ has the value `false` (meaning x_i is absent). The set of clock events according to $\hat{I}$ and the set of all clock events of $\Phi(P)$ are computed as follows:

$$S_{\mathscr{E}c_X}(\hat{I}) = \{I \in \mathscr{E}c_X |\ \forall x_i \in X, (I(x_i) = \hat{I}(\widetilde{x_i}) \wedge \hat{I}(\widehat{x_i}) = \texttt{true}) \vee (I(x_i) = \bot \wedge \hat{I}(\widehat{x_i}) = \texttt{false})\}$$

$$S_{\mathscr{E}c_X}(\Phi(P)) = \bigcup_{\hat{I} \models \Phi(P)} S_{\mathscr{E}c_X}(\hat{I})$$

With a set of clock events $S_{\mathscr{E}c_X}(\Phi(P))$, the concrete clock semantics of $\Phi(P)$ is defined by the following set of clock traces:

$$\Gamma(\Phi(P)) = \{T_c \in \mathscr{T}c_X |\ \forall t, T_c(t) \in S_{\mathscr{E}c_X}(\Phi(P))\}$$

For any subset $X_1 \subseteq X$, the concrete clock semantics of $\Phi(P)$ on X_1 is defined as follows:

$$\Gamma(\Phi(P))_{\setminus X_1} = \{X_1.T_c |\ T_c \in \mathscr{T}c_X \text{ and } \forall t, T_c(t) \in S_{\mathscr{E}c_X}(\Phi(P))\}$$

Definition 20 *Given the abstraction $\Phi(P)$, a property φ defined over the set of clocks $\hat{X}$ is satisfied by $\Phi(P)$ if for any interpretation $\hat{I}$, $\hat{I} \models \Phi(P)$ whenever $\hat{I} \models \varphi$, denoted by $\Phi(P) \models \varphi$.*

To show the soundness of our abstraction, we consider a similar reasoning as in [61]. Our abstraction above is sound in terms of preservation of the clock semantics of the abstracted program P: *if the clock semantics of the abstraction satisfies a property defined over the clocks, then the abstracted program also satisfies this property as stated by the following proposition*. For any property φ which is defined over the set $\hat{X}$, its concretization $\Gamma(\varphi)$ is given by:

$$S_{\mathscr{E}c_X}(\varphi) = \bigcup_{\hat{I} \models \varphi} S_{\mathscr{E}c_X}(\hat{I})$$

$$\Gamma(\varphi) = \{T_c \in \mathscr{T}c_X |\ \forall t, T_c(t) \in S_{\mathscr{E}c_X}(\varphi)\}$$

Proposition 3 *Let P, $\Phi(P)$ be a program and its abstraction, respectively, and φ is a property defined over the clocks. If $\Phi(P) \models \varphi$ then $[[P]]_c \subseteq \Gamma(\varphi)$.*

Proof 6 *The proof of Proposition 3 is done by using Lemma 2. Given a clock trace $T_c \in [[P]]_c$, applying Lemma 2, $T_c \in \Gamma(\Phi(P))$ means that $\forall t, T_c(t) \in S_{\mathscr{E}c_X}(\Phi(P))$. Since $\Phi(P) \models \varphi$, then every interpretation $\hat{I}$ satisfying $\Phi(P)$ also satisfies φ. Thus, any clock event $I \in S_{\mathscr{E}c_X}(\Phi(P))$ is also in $S_{\mathscr{E}c_X}(\varphi)$, meaning that $\forall t, T_c(t) \in S_{\mathscr{E}c_X}(\varphi)$. Therefore, we have $T_c \in \Gamma(\varphi)$.*

Lemma 2 *For all programs P, $[[P]]_c \subseteq \Gamma(\Phi(P))$.*

Proof 7 *We prove it by induction on the structure of program P, meaning that for every primitive operator of the language we show that its clock semantics is a subset of the corresponding concretization.*

- *Stepwise functions: $P : y := f(x_1, ..., x_n)$. First, consider y as numerical signal; following the encoding scheme, we have $\Phi(P) = (\hat{y} \Leftrightarrow \widehat{v_f^k} \Leftrightarrow \widehat{x_1} \Leftrightarrow ... \Leftrightarrow \widehat{x_n}) \wedge (\hat{y} \Rightarrow \tilde{y} = \widetilde{v_f^k})$. For any interpretation $\hat{I}$ such that $\hat{I} \models \Phi(P)$, we have:*

  - *either $\forall i, \hat{y} = \texttt{false}, \widehat{v_f^k} = \texttt{false}$ and $\widehat{x_i} = \texttt{false}$;*
  - *or $\hat{y} = \texttt{true}, \widehat{v_f^k} = \texttt{true}, \forall i. \widehat{x_i} = \texttt{true}$ and $\tilde{y} = \widetilde{v_f^k}$.*

  Then the set of clock events according to $\hat{I}$ is given as follows:

$$S_{\mathscr{E}c_X}(\hat{I}) = \{I \in \mathscr{E}c_{\{y,x_1,...,x_n,v_f^k\}} \mid I(y) = \hat{I}(\tilde{y}) \wedge \forall i, I(x_i) = \hat{I}(\widetilde{x_i}) \text{ if } \hat{I}(\hat{y}) = \hat{I}(\widehat{v_f^k}) = \hat{I}(\widehat{x_i}) = \texttt{true} \text{ and } I(y) = I(x_i) = \bot \text{ if } \hat{I}(\hat{y}) = \hat{I}(\widehat{v_f^k}) = \hat{I}(\widehat{x_i}) = \texttt{false}\}$$

  Let $T_c \in [[P]]_c$ be a clock trace and $t \in \mathbb{N}$ be any instant, then either $\forall i, T_c(t)(y) = T_c(t)(x_i) = \bot$ or $T_c(t)(y) = f(T_c(t)(x_1), T_c(t)(x_2), ..., T_c(t)(x_n))$. We have $T_c(t) \in S_{\mathscr{E}c_X}(\Phi(P))$ for every instant t. Therefore, $T_c \in \Gamma(\Phi(P))$. When y is a Boolean signal, the proof is similar.

- *Delay, sampling, and merging operators: we prove in the same manner.*

- *Composition: $P = P1|P2$. Let $T_c \in [[P]]_c$ be a clock trace, since $X_1.T_c \in [[P_1]]_c, X_2.T_c \in [[P_2]]_c, [[P_1]] \subseteq \Gamma(\Phi(P_1))$ and $[[P_2]]_c \subseteq \Gamma(\Phi(P_2))$, we have $\forall t, T_c(t) \in S_{\mathscr{E}c_X}(\Phi(P_1))$ and $T_c(t) \in S_{\mathscr{E}c_X}(\Phi(P_2))$. That means $\forall t, T_c(t) \in S_{\mathscr{E}c_X}(\Phi(P_1) \wedge \Phi(P_2))$, or $T_c \in \Gamma(\Phi(P))$.*

4.4.3 Definition of correct transformation: Clock refinement

4.4.3.1 Clock refinement

Let $\Phi(A)$ and $\Phi(C)$ be two clock models of programs A and C, to which we refer respectively as a source program and its transformed counterpart produced by the compiler. We denote the sets of all signals in A, C by X_A and X_C, respectively. The corresponding sets of variables which are used to construct the clock models are denoted by $\widehat{X_A}$ and $\widehat{X_C}$.

Consider the finite set of common signals $X = X_A \cap X_C$ and the set of common variables which are used to construct the clock models is $\widehat{X} = \widehat{X_A} \cap \widehat{X_C}$, we say that A and C have the same clock semantics on X if $\Phi(A)$ and $\Phi(C)$ have the same set of concrete restriction clock traces on X:

$$\forall X.T_c.(X.T_c \in \Gamma(\Phi(C))_{\setminus X} \Leftrightarrow X.T_c \in \Gamma(\Phi(A))_{\setminus X}) \tag{4.3}$$

In fact, the compilation makes the transformed program more concrete. For instance, when the SIGNAL compiler performs the "endochronization" which is used to generate the sequential executable code (see Section 3.3.2), the signal with the fastest clock is always present in the generated code. Moreover, compilers perform transformations and optimizations for removing or eliminating some redundant behaviors of the source program (e.g., eliminating subexpressions, trivial clock relations). Additionally, the SIGNAL compiler strengthens the data-flow dependencies expressed in the original program to produce sequential code. Consequently, Requirement (4.3) is too strong to be practical. Hence, we have to relax the requirement as follows:

$$\forall X.T_c.(X.T_c \in \Gamma(\Phi(C))_{\setminus X} \Rightarrow X.T_c \in \Gamma(\Phi(A))_{\setminus X}) \tag{4.4}$$

Requirement (4.4) expresses that every restriction clock trace of $\Phi(C)$ is also a clock trace of $\Phi(A)$ on X, or $\Gamma(\Phi(C))_{\setminus X} \subseteq \Gamma(\Phi(A))_{\setminus X}$. We say that $\Phi(C)$ is a *correct clock transformation* of $\Phi(A)$, or $\Phi(C)$ is a clock refinement of $\Phi(A)$ on X, denoted by $\Phi(C) \sqsubseteq_{clk} \Phi(A)$.

Proposition 4 *The clock refinement is reflexive and transitive:*

- $\forall \Phi(P), \Phi(P) \sqsubseteq_{clk} \Phi(P)$ *on* X_P.
- *Assume that* $X \subseteq X_{P_1}, X \subseteq X_{P_2}$ *and* $X \subseteq X_{P_3}$. *If* $\Phi(P_1) \sqsubseteq_{clk} \Phi(P_2)$ *and* $\Phi(P_2) \sqsubseteq_{clk} \Phi(P_3)$ *on* X, *then* $\Phi(P_1) \sqsubseteq_{clk} \Phi(P_3)$ *on* X.

Proof 8 *Proposition 4 is proved based on the clock refinement definition.*

- *Reflexivity: For every restriction clock trace* $X.T_c$*, we have* $X.T_c \in \Gamma(\Phi(P))_{\backslash X} \Rightarrow X.T_c \in \Gamma(\Phi(P))_{\backslash X}$.

- *Transitivity: For every clock trace* $X.T_c \in \Gamma(\Phi(P_1))_{\backslash X}$*, we have* $X.T_c \in \Gamma(\Phi(P_2))_{\backslash X}$. *Since* $\Phi(P_2) \sqsubseteq_{clk} \Phi(P_3)$ *on* X*, we have* $X.T_c \in \Gamma(\Phi(P_3))_{\backslash X}$*, or* $\Phi(P_1) \sqsubseteq_{clk} \Phi(P_3)$ *on* X.

4.4.3.2 Adaptation to SIGNAL compiler

We will adapt the definition of the above general clock refinement to the case of the SIGNAL compiler to prove the preservation of clock semantics when the compiler transforms one SIGNAL program into the intermediate representation form written in SIGNAL as well. We need to consider the following factors.

A first consideration is that the SIGNAL programs take the inputs from their environment and the register values. Then, they calculate the outputs to react with the environment. In general, the programs can use some local variables to make the output calculations. However, from the outside, the natural observation of the programs is the snapshot of the values of the input and output signals. In our context, it is the snapshot of the presence of the input and output signals. For example, for the program `DEC`, the observation is the tuple of the presence of the signals (FB, N) at a considered instant.

A second consideration is that in the compilation process of the SIGNAL compiler, the local signals in the source program do not necessarily have counterparts in the transformed program. However, all input and output signals are preserved in the transformations and are represented by identical names in the transformed program. Moreover, all signals in the R set are also preserved in the transformations. Therefore, it is natural to choose the snapshot of the presence of the input and output signals to be the observation for the transformed program.

These considerations let us adapt the above definition of clock refinement as follows. Let X_A and X_C be the sets of all signals in the source program A and its counterpart transformed program C. We write X_{IO} to denote the set of common input and output signals. We say that C is correct transformation of A if at any instant, the tuples of values representing the presence of the signals in X_{IO} are the same in both programs. In other words, $\Phi(C) \sqsubseteq_{clk} \Phi(A)$ on X_{IO}.

4.4.4 Proving clock refinement by SMT

We shall propose a method to check the existence of refinement between two clock models with the use of a SMT solver. Let $\Phi(A)$ and $\Phi(C)$ be the clock models of given input and transformed programs. We denote the set of all common input and output signals between A and C by X_{IO}. Our aim is proving that $\Phi(C)$ refines $\Phi(A)$ on X_{IO}. Let $\widehat{X_A}$, $\widehat{X_C}$ and $\widehat{X_{IO}}$ be the set of variables which are used to construct $\Phi(A)$, $\Phi(C)$ and the set of common variables between the two clock models.

For every variable in the clock model $\Phi(C)$ except the common variables in $\widehat{X_{IO}}$, we added "c" as superscript to distinguish them from the variables in the clock model of the input program. The standard way of proving the existence of the clock refinement is based on the following elements:

- The identification of a *variable mapping* that maps the non input, output variables from the clock model $\Phi(A)$ to the non input, output variables in the clock model $\Phi(C)$. We denote the mapping by:

$$\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$$

- The premises of a rule such that if the premises hold, then the conclusion, $\Phi(C)$ refines $\Phi(A)$, is `true`. The premise is presented in Figure 4.3.

For a variable mapping $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$,
Premise. $\forall \hat{I}$ over $\widehat{X_A} \cup \widehat{X_C}.(\hat{I} \models \Phi(C) \Rightarrow \hat{I} \models \Phi(A))$

Conclusion. $\Phi(C) \sqsubseteq_{clk} Phi(A)$ on X_{IO}

Fig. 4.3 Rule CLKREF

The rule CLKREF says that for any interpretation $\hat{I}$ over $\widehat{X_A} \cup \widehat{X_C}$ such that the variable mapping is evaluated to `true`, $\hat{I}$ is a clock configuration of $\Phi(C)$ then it is also a clock model of $\Phi(A)$. Then there exists a clock refinement for $(\Phi(C), \Phi(A))$. The rule CLKREF is sound based on the following theorem.

Theorem 4 *For a variable mapping $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$, if the formula $\Phi(C) \Rightarrow \Phi(A)$ is valid, then $\Phi(C) \sqsubseteq_{clk} \Phi(A)$ on X_{IO}.*

Proof 9 *To prove it, we have to show that for every interpretation $\hat{I}$ over $\hat{X} = \widehat{X_A} \cup \widehat{X_C}$ such that it is evaluated to* `true`. *If $\hat{I} \models (\Phi(C) \Rightarrow \Phi(A))$, then $\Gamma(\Phi(C))_{\setminus X_{IO}} \subseteq \Gamma(\Phi(A))_{\setminus X_{IO}}$. Given*

$X_{IO}.T_c \in \Gamma(\Phi(C))_{\setminus X_{IO}}$, it means that $\forall t, T_c(t) \in S_{\mathscr{E}c_X}(\Phi(C))$. Since for every interpretation $\hat{I}$, $\hat{I} \models \Phi(C)$ implies that $\hat{I} \models \Phi(A)$), thus $S_{\mathscr{E}c_X}(\Phi(C)) \subseteq S_{\mathscr{E}c_X}(\Phi(A))$ under the variable mapping. We get $T_c(t) \in S_{\mathscr{E}c_X}(\Phi(A))$ for every t. Therefore, we have $T_c \in \Gamma(\Phi(A))$.

Consider a variable $x \in \widehat{X_A} \setminus \widehat{X_{IO}}$, the mapping α_x of the variable mapping defines the value of x in the clock model $\Phi(A)$ α-related to the value represented by the clock model $\Phi(C)$. We therefore need to describe the mappings α_x for $x^c \in \widehat{X_C} \setminus \widehat{X_{IO}} = M_{clk} \cup R_{clk} \cup L_{clk}$.

Recall that every state signal s, we introduce a memorization variable $m.s$ in the clock model $\Phi(A)$, and a corresponding a memorization variable $m.s^c$ in the clock model $\Phi(C)$. For example, the state signal N in the equation $ZN := N\$1$ `init` 1, makes use of the memorization variables $m.N$ and $m.N^c$ in the clock models. Therefore, we define the following instance of the α mapping for each state signal s:

$$\tilde{s} \Leftrightarrow \widetilde{s^c} \Rightarrow m.s \Leftrightarrow m.s^c \text{ if } s \text{ is Boolean signal}$$
$$\tilde{s} = \widetilde{s^c} \Rightarrow m.s = m.s^c \text{ if } s \text{ is non-Boolean signal}$$

For example, the mapping for the variables $m.N$ and $m.N^c$ will be given by the formula:

$$\tilde{N} = \widetilde{N^c} \Rightarrow m.N = m.N^c$$

It remains to define the instance of the mapping α for variables $\hat{l}, \tilde{l} \in S_{clk} \cup L_{clk}$ in the clock model $\Phi(A)$ which correspond to the local or state signal named l in the SIGNAL program. In a SIGNAL program, one signal is defined by an equation $l = eq$, if we follow the definitions of all output and local signals in this equation and apply successively substitutions, then we get that the equation is constructed only by the input and state signals. This property is yielded since the SIGNAL program is determinate, meaning that all definitions of signals are defined determinately by the input and state signals, and the SIGNAL compilers rejects all non-determinate program. Equivalently, in the corresponding clock model $\Phi(A)$, the output, state and local variables are determinately defined by the input I and memorization M variables. The definition is written in the clock model in the form $\hat{l} \Leftrightarrow \hat{f} \wedge \tilde{l} = \tilde{f}$, where $\hat{f}$ and $\tilde{f}$ are the formulas which define the clock relations and the values of the signal l in the clock model $\Phi(A)$. Therefore, we define the following instance of the α mapping in the clock model corresponding to each state or local signal l:

$$\hat{l} \Leftrightarrow \hat{f} \wedge \tilde{l} \Leftrightarrow \tilde{f} \text{ if } l \text{ is Boolean signal}$$
$$\hat{l} \Leftrightarrow \hat{f} \wedge \tilde{l} = \tilde{f} \text{ if } l \text{ is non-Boolean signal}$$

For example, the mapping for the variables $\widehat{ZN}$ and $\widetilde{ZN}$ in the clock model $\Phi(DEC)$ corresponding to the local variable ZN in the SIGNAL program DEC will be given by the formula:

$$(\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \tilde{N})) \wedge (m.N_0 = 1)$$

Therefore, the variable mapping $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$ is expressed as the following formula: (1) when the signal is of Boolean type and (2) the signal is of non-Boolean type.

$$\bigwedge_{m.s \in M} \begin{pmatrix} \tilde{s} \Leftrightarrow \widetilde{s^c} \Rightarrow m.s \Leftrightarrow m.s^c & (1) \\ \tilde{s} = \widetilde{s^c} \Rightarrow m.s = m.s^c & (2) \end{pmatrix} \wedge \bigwedge_{\hat{l},\tilde{l} \in S \cup L} \begin{pmatrix} \hat{l} \Leftrightarrow \hat{f} \wedge \tilde{l} \Leftrightarrow \tilde{f} & (1) \\ \hat{l} \Leftrightarrow \hat{f} \wedge \tilde{l} = \tilde{f} & (2) \end{pmatrix}$$

To solve the validity of the formula $(\Phi(C) \Rightarrow \Phi(A))$ in Theorem 4 under the variable mapping, a SMT solver is needed since this formula involves non-Boolean variables and *uninterpreted functions* (using a SAT solver would not be sufficient). A SMT solver decides the satisfiability of arbitrary logic formulas of linear real and integer arithmetic, scalar types, other user-defined data structures, and uninterpreted functions. If the formula belongs to the decidable theory, the solver gives two types of answers: `sat` when the formula has a model (there exists an interpretation that satisfies it); or `unsat`, otherwise. In our case, we will ask the solver to check whether the formula $\neg(\Phi(C) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(A))$ is unsatisfiable, since this formula is unsatisfiable iff $\models (\Phi(C) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(A))$. In our translation validation, the clock models which are constructed from Boolean or numerical variables and uninterpreted functions belong to a part of first-order logic which has a *small model* property according to [28]. The numerical variables are involved only in some implications with *uninterpreted functions* such as

$$(\tilde{x} = \widetilde{x'} \wedge \tilde{y} = \widetilde{y'}) \Rightarrow \widetilde{v^i_\square} = \widetilde{v^j_\square}$$

In addition, the formula is quantifier-free. This means that the check of satisfiability can be established by examining a certain finite cardinality of models. Therefore, the formula can be solved efficiently and significantly improves the scalability of the solver.

4.4.5 Implementation with SMT

We describe the main steps of our approach, and the main techniques which are used to implement them. This implementation can be integrated into the existing POLYCHRONY toolset to prove the preservation of clock semantics.

4.4.5.1 Implementation

Given a source program A, with an unverified compiler, we consider the following compilation process:

1. The compiler takes program A and transforms it.
2. If there is any error (e.g., syntax error), it outputs an `Error`.
3. Otherwise, it outputs the intermediate representation $C = IR(A)$.

These steps can be represented in the following pseudo-code, where $Cp(A)$ is the compilation from the source program A to either compiled program $IR(A)$ or compilation error:

```
1 if (Cp(A) is Error) return Error;
2 else return IR(A);
```

If we associate a verification process which checks that $IR(A)$ refines A with respect to the clock semantics, we obtain the following derived compiler that gives a guarantee of the preservation of clock semantics during the compilation from A to $IR(A)$.

```
1 if (Cp(A) == Error) return Error;
2 else
3 {
4     if (Φ(IR(A)) ⊑clk Φ(A)) return IR(A);
5     else return Error;
6 }
```

We design a validator depicted in Figure 4.4 that takes the source program A and the counterpart compiled program C to construct the corresponding clock models. Then, it establishes the first-order logic formula with the above mapping from C to A as the input of the solver. Finally, in the solving phase, it checks the validity of the formula $(\Phi(C) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(A))$.

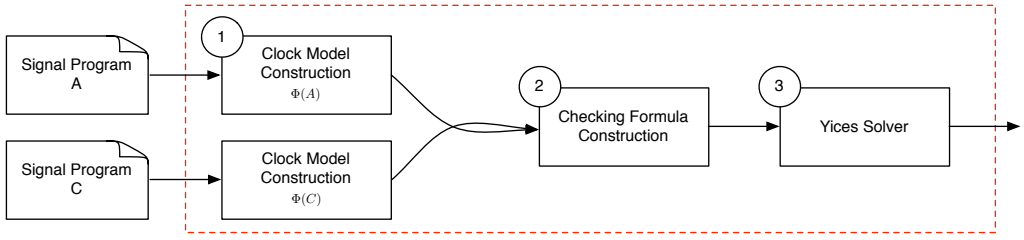

Fig. 4.4 The clock model translation validation

Clock model construction Given the input program A and its transformed programs C, this step will construct the corresponding clock models according to the above encoding scheme. At the end of this step, the clock models of the input and transformed programs are represented as the first-order logic formulas $\Phi(A)$ and $\Phi(C)$, respectively.

Checking formula construction Consider the clock models $\Phi(A)$ and $\Phi(C)$. This step first establishes the variable mapping, then it constructs the formula $(\Phi(C) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(A))$ which is the input of the solving phase for checking its validity.

SMT-based proof We delegate checking the validity of the checking formula to a SMT solver. For our implementation, we consider the YICES solver [51], which is one of the best solvers at the SMT-COMP competition [131].

Let us illustrate the above process on the program DEC in Listing 3.3, and its transformed program DEC_BASIC_TRA in Listing 4.5 at the first phase of the compilation process.

Listing 4.5 DEC_BASIC_TRA in Signal

```
1 (| CLK := CLK_N ^- CLK_FB |)
2  | (| CLK_N := CLK_N ^+ CLK_FB
3     | CLK_N ^= N ^= ZN
4     |)
5  | (| CLK_FB := when (ZN<=1)
6     | CLK_FB ^= FB
7     | CLK_12 := when (not (ZN<=1))
8     |)
9  | (| N := (FB when CLK_FB) default ((ZN-1) when CLK)
10    | ZN := N$1 init 1
11    |)
12 |)
```

In the first step, we will construct the clock models which are presented by the following first-order logic formulas $\Phi(\texttt{DEC})$ and $\Phi(\texttt{DEC\_BASIC\_TRA})$, respectively, where $ZN <= 1, ZN^c <= 1, ZN - 1$ and $ZN^c - 1$ are replaced by the fresh variables $ZN1, ZN1^c, ZN2$ and $ZN2^c$. The uninterpreted function symbols $v^1_{<=}, v^{1c}_{<=}, v^1_-$ and v^{1c}_- are used to encode the step-

wise functions $ZN1 := ZN <= 1, ZN1^c := ZN^c <= 1, ZN2 := ZN - 1$ and $ZN2^c := ZN^c - 1$:

$$
\begin{aligned}
& \Phi(\texttt{DEC}) = \\
& (\widehat{FB} \Leftrightarrow \widehat{ZN1} \wedge \widetilde{ZN1}) \\
\wedge\ & (\widehat{ZN1} \Leftrightarrow \widehat{v^1_{<=}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN1} \Rightarrow (\widetilde{ZN1} = \widetilde{v^1_{<=}})) \\
\wedge\ & (\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \tilde{N})) \wedge (m.N_0 = 1) \\
\wedge\ & (\widehat{N} \Leftrightarrow \widehat{FB} \vee \widehat{ZN2}) \wedge (\widehat{N} \Rightarrow ((\widehat{FB} \wedge \tilde{N} = \overline{FB}) \vee (\neg\widehat{FB} \wedge \tilde{N} = \widetilde{ZN2}))) \\
\wedge\ & (\widehat{ZN2} \Leftrightarrow \widehat{v^1_-} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN2} \Rightarrow (\widetilde{ZN2} = \widetilde{v^1_-}))
\end{aligned}
$$

$$
\begin{aligned}
& \Phi(\texttt{DEC\_BASIC\_TRA}) = \\
& (\widehat{CLK^c} \Leftrightarrow \widehat{CLK\_N^c} \wedge \neg\widehat{CLK\_FB^c}) \wedge (\widehat{CLK^c} \Rightarrow \widetilde{CLK^c}) \\
\wedge\ & (\widehat{CLK\_N^c} \Leftrightarrow \widehat{CLK\_N^c} \vee \widehat{CLK\_FB^c}) \wedge (\widehat{CLK\_N^c} \Rightarrow \widetilde{CLK\_N^c}) \\
\wedge\ & (\widehat{CLK\_N^c} \Leftrightarrow \widehat{N} \Leftrightarrow \widehat{ZN^c}) \\
\wedge\ & (\widehat{N} \Leftrightarrow \widehat{FB} \wedge \widehat{CLK\_FB^c} \wedge \widetilde{CLK\_FB^c} \vee \widehat{ZN2^c} \wedge \widehat{CLK^c} \wedge \widetilde{CLK^c}) \\
\wedge\ & (\widehat{N} \Rightarrow (\widehat{FB} \wedge \widehat{CLK\_FB^c} \wedge \widetilde{CLK\_FB^c} \wedge \tilde{N} = \widetilde{FB} \\
& \vee\neg(\widehat{FB} \wedge \widehat{CLK\_FB^c} \wedge \widetilde{CLK\_FB^c}) \wedge \tilde{N} = \widetilde{ZN2^c})) \\
\wedge\ & (\widehat{ZN2^c} \Leftrightarrow \widehat{v^{1c}_-} \Leftrightarrow \widehat{ZN^c}) \wedge (\widehat{ZN2^c} \Rightarrow (\widetilde{ZN2^c} = \widetilde{v^{1c}_-})) \\
\wedge\ & (\widehat{ZN^c} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN^c} \Rightarrow (\widetilde{ZN^c} = m.N \wedge m.N' = \tilde{N})) \wedge (m.N_0 = 1 \\
\wedge\ & (\widehat{CLK\_FB^c} \Leftrightarrow \widehat{ZN1^c} \wedge \widetilde{ZN1^c}) \wedge (\widehat{CLK\_FB^c} \Rightarrow \widetilde{CLK\_FB^c}) \\
\wedge\ & (\widehat{ZN1^c} \Leftrightarrow \widehat{v^{1c}_{<=}} \Leftrightarrow \widehat{ZN^c}) \wedge (\widehat{ZN1^c} \Rightarrow (\widetilde{ZN1^c} = \widetilde{v^{1c}_{<=}})) \\
\wedge\ & (\widehat{CLK\_FB^c} \Leftrightarrow \widehat{FB}) \\
\wedge\ & (\widehat{CLK\_12^c} \Leftrightarrow \widehat{ZN1^c} \wedge \neg\widetilde{ZN1^c}) \wedge (\widehat{CLK\_12^c} \Rightarrow \widetilde{CLK\_12^c})
\end{aligned}
$$

In the second step, checking formula construction, our tool will establish the variable mapping $\widehat{X_{\texttt{DEC}}} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_{\texttt{DEC\_BASIC\_TRA}}} \setminus \widehat{X_{IO}})$ as follows:

$$
\begin{aligned}
& (\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \tilde{N})) \wedge (m.N_0 = 1) \\
\wedge\ & (\widehat{ZN1} \Leftrightarrow \widehat{v^1_{<=}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN1} \Rightarrow (\widetilde{ZN1} = \widetilde{v^1_{<=}})) \\
\wedge\ & (\widehat{ZN2} \Leftrightarrow \widehat{v^1_-} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN2} \Rightarrow (\widetilde{ZN2} = \widetilde{v^1_-}))
\end{aligned}
$$

In the third step, we delegate the checking validity of the formula $(\Phi(\texttt{DEC\_BASIC\_TRA}) \wedge \widehat{X_{\texttt{DEC}}} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_{\texttt{DEC\_BASIC\_TRA}}} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(\texttt{DEC}))$, named φ, to the SMT solver under the logical context defined by the variable mapping and the following assertions:

$$
\begin{aligned}
(\widetilde{ZN} = \widetilde{ZN^c} \wedge 1 = 1) &\Rightarrow \widetilde{v^1_{<=}} = \widetilde{v^{1c}_{<=}} \\
(\widetilde{ZN} = \widetilde{ZN^c} \wedge 1 = 1) &\Rightarrow \widetilde{v^1_-} = \widetilde{v^{1c}_-}
\end{aligned}
$$

With the YICES solver, we will get `unsat` when checking the satisfiability of $\neg\varphi$, which means that φ is valid. Thus, we can conclude that the transformation is correct.

4.4.5.2 Constant clock

We describe the additional features that have to be considered when the translation validation is applied on real SIGNAL programs. In SIGNAL, the occurrence of constants is allowed to designate a constant signal (i.e., a signal with a constant value). However, each occurrence of a constant has a particular clock since the corresponding signal is hidden, and this clock is determined by the context where the constant is used, called *context clock*. This makes our abstraction for SIGNAL operators above invalid in case a constant signal is used. In consequence of that, we have to provide new definitions of the abstraction when the operators use a constant signal (`cst` denotes a constant):

Stepwise functions

- $\phi(y := cst) = \hat{y} \Rightarrow (\tilde{y} \Leftrightarrow cst)$ if y is Boolean signal.
- $\phi(y := cst) = \hat{y} \Rightarrow (\tilde{y} \Leftrightarrow cst)$ if y is non-Boolean signal, where cst is an interpreted function.
- $\phi(y := x \text{ and } cst) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \tilde{x} \wedge cst))$.
- $\phi(y := x \text{ or } cst) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \tilde{x} \vee cst))$.
- $\phi(y := x \,\square\, cst) = (\hat{y} \Leftrightarrow \widehat{v^i_\square} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} = \widetilde{v^i_\square}))$.

Merge

$$\phi(y := x \texttt{ default } cst) = (\hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{y} \Rightarrow (\hat{x} \wedge \tilde{y} = \tilde{x} \vee \neg\hat{x} \wedge \tilde{y} = cst))$$
$$\phi(y := cst \texttt{ default } x) = (\hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{y} \Rightarrow (\hat{y} \wedge \tilde{y} = cst \vee \neg\hat{y} \wedge \tilde{y} = \tilde{x}))$$

Sampling

$$\phi(y := x \texttt{ when true}) = (\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{y} \Rightarrow (\tilde{y} = \tilde{x}))$$
$$\phi(y := x \texttt{ when false}) = \hat{y} \Leftrightarrow \texttt{ false}$$
$$\phi(y := cst \texttt{ when } b) = (\hat{y} \Leftrightarrow (\hat{b} \wedge \tilde{b})) \wedge (\hat{y} \Rightarrow (\tilde{y} = cst))$$

4.4.6 Detected bugs

So far, our validator has revealed two previously unknown bugs in the compilation of the SIGNAL compiler. One of them is related to the multiple constraints of clocks. Another one is a syntax error of generated C code from a SIGNAL program in which a constant signal is used.

The first problem was introduced when multiple constraints condition a clock such as in the following segment of SIGNAL program and its clock calculation part in transformed programs:

```
1  /* P.SIG */
2  | x ^= when (y <= 9)
3  | x ^= when (y >= 1)
4  /* P_BASIC_TRA.SIG */
5  ...
6  | CLK_x := when (y <= 9)
7  | CLK := when (y >= 1)
8  | CLK_x ^= CLK
9  | CLK ^= XZX_24
10 ...
11 /* P_BOOL_TRA.SIG */
12 ...
13 | when Tick ^= C_z ^= C_CLK
14 | when C_z ^= x ^= z
15 | C_z := y <= 9
16 | C_CLK := y >= 1
17 ...
```

In the transformed counterpart P_BASIC_TRA, the introduction of signal XZX_24 and the synchronization between CLK and XZX_24 cause the incorrect specification of clocks (in program P_BASIC_TRA, the signal x might be absent when XZX_24 is absent, which is not the case in the source program P, nor in P_BOOL_TRA). This bug was caught by our validator when it found that $\Phi(\texttt{P\_BOOL\_TRA}) \not\sqsubseteq_{clk} \Phi(\texttt{P\_BASIC\_TRA})$. In addition, signal XZX_24 is introduced without declaration, which makes a syntax error in P_BASIC_TRA.

The second problem detected was not found by the translation validation but was indirectly discovered when trying to apply it. It occurred in a SIGNAL program in which a *merge* operator with a constant signal was used, such as $y := 1$ `default` x. In this case, the code generation phase of the compiler dealt wrongly with the *clock context* of a constant signal by introducing a syntax error in the generated C code. The bug and its fix are given by:

```
1 /* Version with bug */
2 if (C_y)
3 {
4    y = 1; else y = x;
5    w_ClockError_y(y);
6 }
7 /* Version without bug */
8 if (C_y)
9 {
10   if (C_y) y = 1; else y = x;
11   w_ClockError_y(y);
12 }
```

4.5 Discussion

Beside the works which have adopted the translation validation approach in verification of a compiler as discussed in Chapter 2, the static analysis of SIGNAL programs for efficient code generation [61] can be considered as our closest related work.

The common semantics framework which are used to construct the translation validation of *clock models* in this chapter is inspired by the interval-Boolean abstraction of [61]. The objective of this paper was to propose a method to make the code generated by the Signal compiler more efficient by detecting and removing the dead-code segments (e.g., segment of code to compute a data-flow which is always absent). The technique allows to detect empty clocks, mutual exclusion of clocks, or clock inclusions, by reasoning on an interval-Boolean model using an SMT solver. The choice of SMT solvers is explained by the fact that it is more judicious than the *Interval Decision Diagram* (IDD) technique used in [63]. In IDDs, intervals are defined on integers. To deal with other data types, IDDs require an encoding into integers. Using SMT solving, the story is different, since the solvers support a large range of algebras. From a practical point of view, SMT solving is also easier to integrate in the current Signal compiler. In our work, we used an interval-Boolean abstraction to represent the condition of dependencies among signals to make a more efficient deadlock detection technique.

On the other hand, Pouzet et al. [22] introduce a generic machine-based intermediate presentation to describe the transition functions in the modular compilation of SCADE/LUSTRE. The formalization of the intermediate presentation appears as a fundamental need in order to develop a certified compiler using a proof assistant. However, as we mentioned above, this approach yields a situation where any change of the compiler requires redoing

the proof. Moreover, a realistic compiler is in general a much bigger and more difficult object to verify than a single translation function.

The present chapter provides a proof of correctness of a multi-clocked synchronous programming language compiler for clock semantics preservation and applies this approach to the synchronous data-flow language SIGNAL compiler. We have presented two approaches based on model checking and the use of SMT solving to prove the preservation of clock semantics during the compilation. The clock semantics of a given source program and its transformed program are represented as PDSs over the finite field modulo 3, $\mathbb{Z}/_3\mathbb{Z}$. A refinement relation between the source program and its compiled program is used to express the preservation. A proof based on the simulation technique is presented to establish the existence of the refinement relation. In the second approach based on the use of SMT solver, the clock semantics is represented as a clock model. A refinement relation between two clock models is used to express the preservation, which is checked by using a SMT solver.

A compilation phase, followed by the refinement verification process, guarantees that the clock semantics of the source program is preserved in the compiled program. We have proposed methods to implement and integrate our verification process within the POLYCHRONY toolset by extending the functionality of the existing model checker SIGALI, and the use of SMT solver to prove the correctness of the first two phases of the SIGNAL compiler.

Thanks to this experiment, we observed that the approach based on model checking suffers from the increasing state-space when it deals with large programs (the programs contain a huge amount of variables). The number of states grows exponentially with the number of program variables. On the contrary, in the approach based on the use of SMT solver, the clock semantics is presented as a first-order logic formula over Boolean variables and *uninterpreted functions*. Thanks to the efficiency of the SMT implementation, this approach can deal with programs whose number of variables is very big.

CHAPTER 5

TRANSLATION VALIDATION OF SDDG

In this chapter, we describe how the preservation of data dependencies among signals in the compilation of SIGNAL can be proved by showing that for every pair of signals x and y in the source program, if there exists a data dependency between x and y, then this dependency is also in the compiled program. The data dependencies among signals in the program are represented by a formal representation, called *Synchronous Data-flow Dependency Graph* (SDDG).

A SDDG is a labeled directed graph in which each node is a signal or a clock, each edge from a node to another node represents the dependency between nodes that is labeled by a condition, called *clock constraint*. Therefore, a dependency between two signals is conditioned: the dependency is effective whenever the condition holds. For instance, the basic process $y := x$ `when` b specifies that at any instant such that x is present, b is present, and b holds the value `true`, y cannot be set before the evaluation of x. We can use a Boolean expression $\hat{x} \wedge \hat{b} \wedge \tilde{b}$ to encode the fact that x is present, b is present, and b holds the value `true`, where $\hat{x}, \hat{b}, \tilde{b}$ are Boolean variables. Hence, the value of y depends on the current value of x whenever the condition $\hat{x} \wedge \hat{b} \wedge \tilde{b}$ is satisfied. Given two SDDGs of the source and compiled programs, a *refinement* relation between them is formally defined which expresses the semantic preservation of data dependency. We delegate checking the refinement to a SMT solver.

Figure 5.1 shows our validation and its integration into the SIGNAL compilation process. If the validator points out that there does not exit a refinement then a "compiler bug" message is emitted. Otherwise, the compiler continues its work.

The remainder of this chapter is organized as follows. The definition of synchronous data-flow dependency graph and the encoding scheme from SIGNAL programs into SDDGs are studied in Section 5.2. The translation validation of SDDGs is described in Section 5.3.

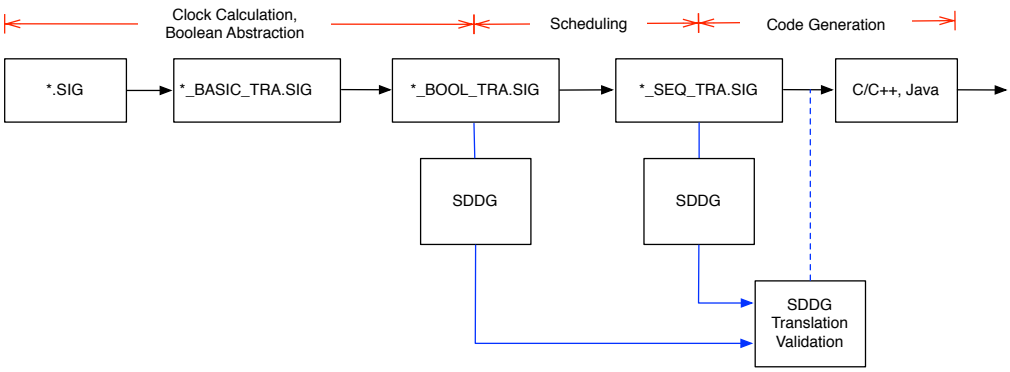

Fig. 5.1 Translation validation of SDDG

We also address the mechanism of the verification process, the application of the verification process to the SIGNAL compiler, and its integration in the POLYCHRONY toolset [77]. Section 5.4 introduces an application of SDDG to build a precise deadlock detection for SIGNAL compiler. We present some related works, conclusion and future research in Section 5.5.

5.1 The data dependency analysis in SIGNAL compiler

The second phase of the SIGNAL compiler, *static scheduling*, aims to make the precise determination of dependencies among signals in programs. The analysis of such dependencies provides some important functions of the compiler. The first function is to detect that the program does not contain any cyclic definition, called *deadlock detection*. The deadlock detection is studied in detail in Section 5.4.1. The other function is to statically calculate the scheduling for a multi-processor implementation.

In order to do the above functions, the SIGNAL compiler constructs a graph which represents the data dependencies among signals, called *Graph of Conditional Dependencies* (GCD) as described in [13]. For any pair of signals (x, y), it specifies under what condition the signal x depends instantly on the signal y.

To detect a cyclic definition, for a cycle in the graph, the compiler computes the conjunction of all the conditions associated with this cycle, and checks that the conjunction is identically false.

The scheduling can be statically calculated by defining subgraphs of the GCD that maybe distributed on different processors. For instance, the computation of two output signals x and y can be distributed on two different cores of a single processor or on two different processors if there is no data dependencies between them.

	Dependency	Encoding in GCD
x	$C_x \xrightarrow{C_x} x$	$x^2 \xrightarrow{x^2} x$
c (Boolean signal)	$c \xrightarrow{[c]} [c]$, $c \xrightarrow{[\neg c]} [\neg c]$	$c \xrightarrow{-c-c^2} (-c-c^2)$ $c \xrightarrow{c-c^2} (c-c^2)$
$x \xrightarrow{c} y$	$[c] \xrightarrow{[c]} y$	$(-c-c^2) \xrightarrow{-c-c^2} y$
$y := f(x_1,...,x_n)$	$x_1 \xrightarrow{C_y} y \ ... \ x_n \xrightarrow{C_y} y$	$x_1 \xrightarrow{y^2} y \ ... \ x_n \xrightarrow{y^2} y$
$y := x\$1$ `init` a		
$y := x$ `when` b	$x \xrightarrow{C_y} y$, $b \xrightarrow{C_y} C_y$	$x \xrightarrow{y^2} y$, $b \xrightarrow{y^2} y^2$
$y := x$ `default` z	$x \xrightarrow{C_x} y$, $z \xrightarrow{C_z \setminus C_x} y$	$x \xrightarrow{x^2} y$, $z \xrightarrow{z^2(1-x^2)} y$

Table 5.1 The implicit dependencies and their encoding in GCD

We first recall the analysis of implicit dependencies among signals of the primitive operators as in Section 3.3.2. We use $x \xrightarrow{c} y$ to denote the fact that there is a dependency between y and x at the condition c. In particular, the following dependencies apply implicitly.

- Any signal is preceded by its clock.
- For a Boolean signal c, $[c]$ and $[\neg c]$ depend on c.
- Any dependency $x \xrightarrow{c} y$ implies implicitly a dependency $[c] \xrightarrow{[c]} y$.

Then, we have the implicit dependencies among signals for the core language that are given in Table 5.1. As an example, for the basic process $y := x$ `when` b, the signal y depends on the signal x whenever y is present. The clock C_y depends on the Boolean signal b whenever y is present.

A GCD calculated by the SIGNAL compiler is a labeled directed graph, in which the vertices are the signals or clock variables. The edges indicate the data dependencies among signals and clock variables, and the labels are clocks which are polynomials whose coefficients range over the finite field of integers modulo 3, $\mathbb{Z}/3\mathbb{Z}$. These clocks represent the conditions at which the dependencies are valid.

For example, we consider the program `DEC` in Section 3.3.2 which emits a sequence of values $FB, FB-1, ..., 2, 1$, from each value of a positive integer signal FB coming from its environment. The data dependencies among the signals FB, ZN and N can be represented by the graph in Figure 5.2, where we introduce two fresh variables $ZN1$ and $ZN2$ to replace

the expressions $ZN <= 1$ and $ZN - 1$, respectively. The clocks which label the edges in the graph are encoded as polynomials over $\mathbb{Z}/_3\mathbb{Z}$ related as follows.

$$\begin{aligned} ZN1^2 = ZN2^2 = ZN^2 &= N^2 \\ FB^2 &= -ZN1 - ZN1^2 \\ N^2 &= FB^2 + (1 - FB^2)ZN2^2 \end{aligned}$$

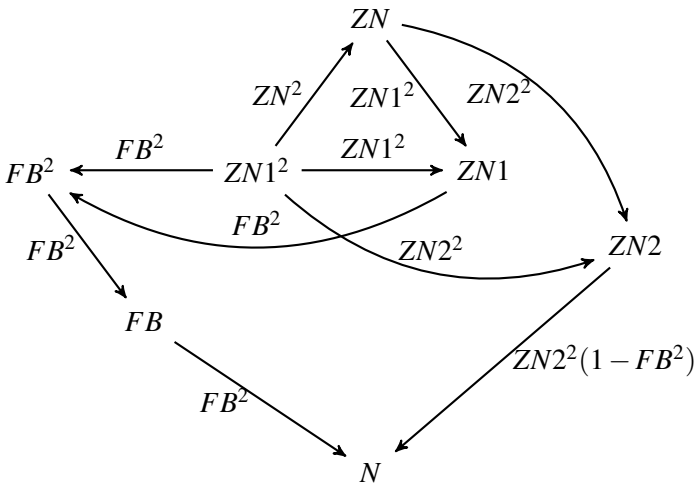

Fig. 5.2 The GCD of DEC

5.2 Synchronous data-flow dependency graph

We will describe the data dependencies among signals in terms of SDDGs. In presenting the construction of a SDDG below, we first explain that a usual *Data Dependency Graph* (DDG) is not sufficient to represent the dependencies among signals in a polychronous program. Next, we show how clock constraints which are represented as first-order logic formulas as in Section 4.4.1 can be used to express the conditional dependencies. Let us consider again the *sampling* operator, $y := x$ `when` b, the dependency of y on x is conditioned by the formula $\hat{x} \wedge \hat{b} \wedge \tilde{b}$. That means that at a given instant, y depends on x only when this formula is evaluated to be `true`.

5.2.1 Data dependency graphs

As in [6], a DDG is a directed graph which contains nodes that represent locations of *definitions* and *uses* of variables in *basic blocks*, and edges that represent data dependencies between nodes. Once the DDG is constructed, the dependencies among variables are fixed. They do not vary along the time, in other words, DDGs can only be used to represent the data dependencies which are static in terms of the time.

Considering the pseudo-code of the following program `Sum`. Figure 5.3 partially shows its DDG (the figure shows only the data dependencies that are related to variable *i*). Data dependency edges are depicted by dotted lines which are added to the *Control Flow Graph* (CFG) [41, 111], and labeled by the name of the variable that creates the dependency. Node numbers in the CFG correspond to statement numbers in the program (we treat each statement as a basic block). Each node that represents a transfer of control (e.g., node 5) has two edges with labels `true` and `false`, all others are unlabeled.

```
1 Program Sum {
2    read(n);
3    i = 1;
4    sum = 0;
5    while (i <= n) {
6       sum = sum + i;
7       i = i + 1;
8    }
9    write(sum);
10 }
```

5.2.2 SIGNAL program as synchronous data-flow dependency graph

Data dependency graphs would not really represent the data dependencys of a SIGNAL program since the presence of signals may vary along time (which is defined by their clock), in consequence, the dependencies among signals in the program are not static, they also vary. To deal with that, the dependencies must be conditioned by the clocks at which the dependencies are effective.

To illustrate the definition of SDDGs, we consider a process which involves the basic process corresponding to the *merge* operator:

```
1 (|
2  | x := expression
3  | z := expression
```

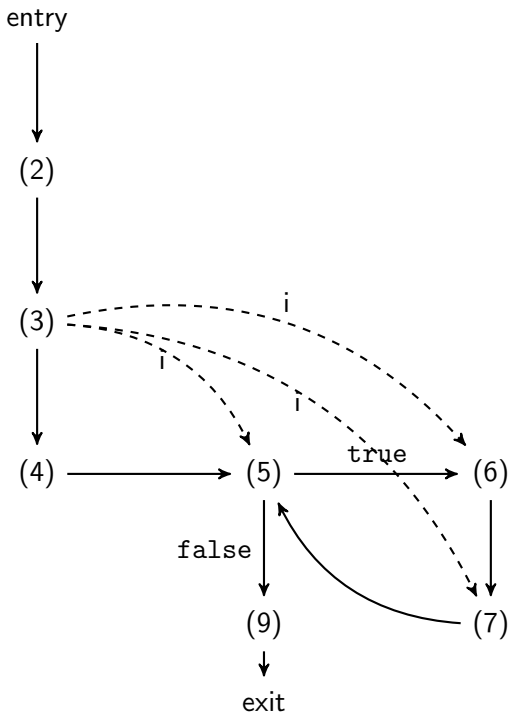

Fig. 5.3 CFG for Sum, with data dependency edges for i (dotted lines)

```
4  | ...
5  | y := x default z
6  |)
```

The statements (2), (3) and (5) represent the expressions defining the signal x, z and y, respectively. The signal x is defined at statement 2 and is fetched at statement 5 if the signal x is present. Considering the basic process $y := x$ `default` z (and the clock constraints between signals), the "valid" states are: x is present and y is present; or x is absent, z is present, and y is present; or x, y and z are absent. They can be expressed by $\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$ in our abstraction as in Section 4.4.1. According to the valid states of the signals, the different data dependencies among signals are depicted in Figure 5.4, where the labels represent the conditions at which the dependencies are effective. For instance, when $\hat{x} =$ `true`, y is de-

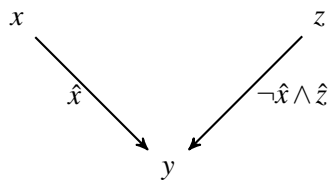

Fig. 5.4 The SDDG of merge operator

fined by x; otherwise it is defined by z when $\hat{x} = \texttt{false}$ and $\hat{z} = \texttt{true}$. The graph has the following property: an edge cannot exist if one of its extremity nodes is not present (or the corresponding signal holds no value). In our example, this property can be translated in our abstraction of clock semantics as:

$$\hat{x} \Rightarrow \hat{y} \wedge \hat{x} \text{ and } \neg\hat{x} \wedge \hat{z} \Rightarrow \hat{y} \wedge \hat{z}$$

A SDDG for a given program is a labeled directed graph in which each node is a signal or clock variable and each edge represents the dependency between nodes. Each edge is labeled by a first-order logic formula that represents the clock at which the dependency between the extremity nodes is effective. Formally, a SDDG is defined as follows:

Definition 21 (SDDG) *A* SDDG *associated with a* SIGNAL *program is a labeled directed graph* $G = \langle N, E, I, O, C, m_N, m_E \rangle$*, where:*

- N *is a finite set of nodes. Each node is a signal or a clock variable.*
- $E \subseteq N \times N$ *is the set of edges. They describe the data dependencies among signals and clock variables in the program. Each edge is labeled by a clock constraint that is a first-order logic formula in our abstraction.*
- $I \subseteq N$ *is the set of input nodes. They are the set of input signals.*
- $O \subseteq N$ *is the set of output nodes. They are the set of output signals.*
- C *is the set of conditions, called clock constraints. The conditions are encoded as the expressions of clocks in the abstraction in Section 4.4.1.*
- $m_N : N \longrightarrow C$ *is a mapping labeling each node with a clock contraint. It defines the existence condition of a node.*
- $m_E : E \longrightarrow C$ *is a mapping labeling each edge with a clock constraint. It defines the existence condition of an edge.*

In contrast with DDG, the clock labeling in SDDG provides a dynamic dependency feature. This clock labelling imposes two properties which are implicit:

- An edge cannot exist if one of its two extremity nodes does not exist. This property can be translated in our abstraction as:

$$\forall (x,y) \in E, m_E(x,y) \Leftarrow (m_N(x) \wedge m_N(y))$$

Operators	Encoding in SDDG
x	$\hat{x} \xrightarrow{\hat{x}} x, m_N(\hat{x}) = \hat{x}, m_N(x) = \hat{x}$
c (Boolean signal)	$c \xrightarrow{[c]} [c], m_N(c) = \hat{c}, m_N([c]) = [c], c \xrightarrow{[\neg c]} [\neg c], m_N(c) = \hat{c}$ $m_N([\neg c]) = [\neg c]$
$x \xrightarrow{c} y$	$[c] \xrightarrow{[c]} y, m_N([c]) = [c], m_N(y) = \hat{y}$
$y := f(x_1, ..., x_n)$	$x_1 \xrightarrow{\hat{y}} y \ ... \ x_n \xrightarrow{\hat{y}} y, m_N(x_i) = \widehat{x_i}, m_N(y) = \hat{y}, i = 1, ..., n$
$y := x\$1$ `init` a	$m_N(x) = \hat{x}, m_N(y) = \hat{y}$
$y := x$ `when` b	$x \xrightarrow{\hat{y}} y, m_N(x) = \hat{x}, m_N(y) = \hat{y}, b \xrightarrow{\hat{y}} \hat{y}, m_N(b) = \hat{b}, m_N(\hat{y}) = \hat{y}$
$y := x$ `default` z	$x \xrightarrow{\hat{x}} y, m_N(x) = \hat{x}, m_N(y) = \hat{y}, z \xrightarrow{\hat{z} \wedge \neg \hat{x}} y, m_N(z) = \hat{z}, m_N(y) = \hat{y}$

Table 5.2 The dependencies of the core language

- A cycle of dependencies does not stand for a deadlock if the conjunction of all the conditions associated with the cycle is identically false. It can be expressed as:

$$x_1, ..., x_n, x_1 \text{ is not a deadlock if } m_E(x_1, x_2) \wedge m_E(x_2, x_3) \wedge ... \wedge m_E(x_n, x_1) \text{ is } \texttt{false}$$

We denote the fact that there exists a dependency between two nodes x and y at a clock constraint $m_E(x, y) = \hat{c}$ by $x \xrightarrow{\hat{c}} y$. Then, a *dependency path* from x to y is any set of nodes $s = \langle x_0, x_1, ..., x_k \rangle$ such that x_{i+1} depends on x_i for all i from 0 to $k-1$. The clock constraint at which the dependency path is effective is $\widehat{c} = \bigwedge_{i=0}^{k-1} \widehat{c_i}$, where $\widehat{c_i} = m_E(x_i, x_{i+1})$. When the number of nodes is two, the path is an edge. We also write $x = x_0 \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{k-1}}} x_k = y$ to denote a path from x to y.

In Table 5.2, we construct the dependencies among signals for the core language, where the sub clocks $[c]$ and $[\neg c]$ are encoded as $\hat{c} \wedge \tilde{c}$ and $\hat{c} \wedge \neg \tilde{c}$, respectively, in our abstraction. The edges are labeled by clocks which are represented by a first-order formula. For instance, the basic process of the primitive operator *sampling* satisfies that $\hat{y} \Rightarrow \hat{x} \wedge \hat{y}$ and $\hat{y} \Rightarrow \hat{b} \wedge \hat{y}$.

We assume that all considered SIGNAL programs are written with the primitive operators, meaning that derived operators are replaced by their definition with primitive ones, and there are no nested operators (these nested operators can be broken by using fresh signals). Following the above construction rules, we can obtain the SDDG in Figure 5.5 for the program `DEC` in Section 3.3.2, where we introduce two fresh variables $ZN1$ and $ZN2$ to replace the expressions $ZN <= 1$ and $ZN - 1$, respectively. Note that we omit some parts of the graph such as the signal N depends on its clock $\widehat{N}$. The clocks which label the edges in the

graph are encoded as the first-order formulas given as follows.

$$\widehat{ZN1} = \widehat{ZN2} = \widehat{ZN} = \widehat{N}$$
$$\widehat{FB} = \widehat{ZN1} \wedge \widetilde{ZN1}$$
$$\widehat{N} = \widehat{FB} \vee \widehat{ZN2}$$

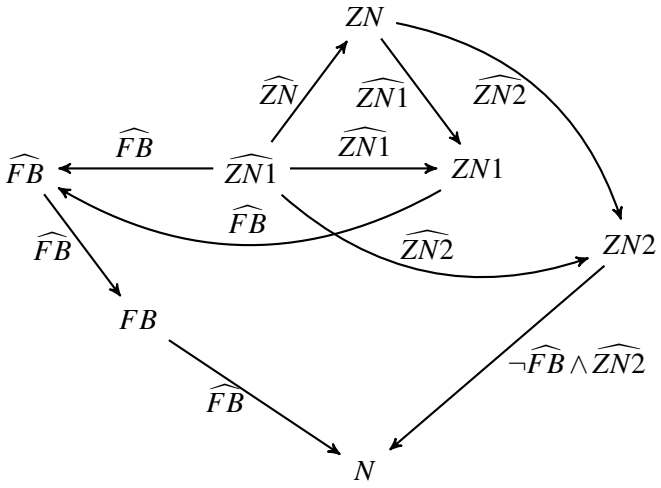

Fig. 5.5 The SDDG of DEC

5.3 Translation validation of SDDG

In this section, we adopt the translation validation approach to prove the correctness of the compiler in the *static scheduling* phase of the compilation process. Given two SDDGs, we first formalize the notion of "correct implementation" as a *dependency refinement* relation. This refinement expresses the semantic preservation of data dependencies. Then, we propose a method to implement our verification framework by the use of a SMT solver for checking the existence of the above refinement relation.

5.3.1 Definition of correct implementation: Dependency refinement

Let SDDG(A) = $\langle N_A, E_A, I_A, O_A, C_A, m_{N_A}, m_{E_A} \rangle$ and SDDG(C) = $\langle N_C, E_C, I_C, O_C, C_C, m_{N_C}, m_{E_C} \rangle$, to which we refer respectively as the data dependency representations of the source program and its transformed counterpart produced by the SIGNAL compiler. Let x and y be two signals in both programs A and C. A dependency path from the signal x to the signal y in

SDDG(C) is a *reinforcement* of the dependency path from x to y in SDDG(A) if at any instant t, the dependency path in SDDG(A) is effective (meaning that the conjunction of all the conditions associated with the path is evaluated to be `true` at t), then the corresponding dependency path in SDDG(C) is also effective. The formal definition of reinforcement is given as follows.

Definition 22 (Reinforcement) *Let $dp_1 = \langle x_0, x_1, ..., x_n \rangle$ and $dp_2 = \langle x'_0, x'_1, ..., x'_m \rangle$ be two dependency paths in* SDDG(A) *and* SDDG(C)*, respectively, where $x = x_0 = x'_0, y = x_n = x'_m$ and $\widehat{c_i}, \widehat{c'_j}$ denote the clock constraints $m_{E_A}(x_i, x_{i+1})$ and $m_{E_C}(x'_j, x'_{j+1})$. It is said that dp_2 is a reinforcement of dp_1 iff the following formula is valid.*

$$\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j}$$

We write $dp_2 \preceq_{dep} dp_1$ to denote the fact that dp_2 is a reinforcement of dp_1. The assertion $\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j}$ indicates that if the dependency path dp_1 in SDDG(A) is effective at any instant, then the dependency path dp_2 in SDDG(C) is also effective. In the special case when $m = n = 1$, $x \xrightarrow{\widehat{c'_0}} y$ is a reinforcement of $x \xrightarrow{\widehat{c_0}} y$ iff $\models \widehat{c_0} \Rightarrow \widehat{c'_0}$.

Consider a dependency path from x to y in a SDDG graph. Assume that there exists a path from y to x, that makes a dependency cycle between x and y. We say that such cycle is a deadlock iff the dependencies of x to y and vice-versa are effective at the same time. The formal definition of deadlock is given as follows.

Definition 23 (Deadlock) *Let $dp = \langle x_0, x_1, ..., x_n, x_0 \rangle$ be a cycle in a* SDDG *graph. The dependency cycle dp stands for a deadlock if the conjunction of all the clock constraints associated with the cycle is satisfiable, meaning that there exists some interpretation that makes the conjunction formula $m_E(x_0, x_1) \wedge m_E(x_1, x_2) \wedge ... \wedge m_E(x_n, x_0)$* `true`.

Obviously, a dependency cycle does not stand for a deadlock if the conjunction of all the clock constraints associated with the cycle, in which the dependencies are effective, is identically `false`. That means the dependencies of the cycle cannot be present at the same time. It can be expressed as:

$$M \not\models \bigwedge_{i=0}^{n} \widehat{c_i}, \text{ where } \widehat{c_i} \text{ is the clock constraint associated with the cycle.}$$

It indicates that there is no interpretation that makes the conjunction of all the clock constraints associated with the cycle `true`. Based on the above definition of deadlock, a SDDG is deadlock-free if every dependency cycle in the graph does not stand for a deadlock.

Definition 24 (Deadlock-consistent) *Let dp_1 and dp_2 be two dependency paths from the signal x to the signal y in* SDDG(A) *and* SDDG(C)*, respectively. The dependency path dp_2 is deadlock-consistent with dp_1 if the following condition is satisfied: if every dependency cycle between x and y in* SDDG(A) *is not a deadlock then all dependency cycles between x and y in* SDDG(C) *are not deadlocks.*

Let $dp_1 = \langle x_0, x_1, ..., x_n \rangle$ and $dp_2 = \langle x'_0, x'_1, ..., x'_m \rangle$ be two dependency paths in SDDG(A) and SDDG(C), respectively, where $x = x_0 = x'_0, y = x_n = x'_m$ and $\widehat{c_i}, \widehat{c'_j}$ denote the clock constraints $m_E(x_i, x_{i+1})$ and $m_E(x'_j, x'_{j+1})$. The Definition 24 can be expressed as follows. For any dependency path $dp_1^{inv} = \langle x_0^{inv}, x_1^{inv}, ..., x_p^{inv} \rangle$, where $x_0^{inv} = y, x_p^{inv} = x$, that forms a cycle between x and y in SDDG(A), then for every dependency path $dp_2^{inv} = \langle x_0^{inv'}, x_1^{inv'}, ..., x_q^{inv'} \rangle$, where $x_0^{inv'} = y, x_q^{inv'} = x$, that forms a cycle between x and y in SDDG(C), it satisfies:

$$\models (\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{c_j^{inv}}) \Leftrightarrow \texttt{false} \Rightarrow (\bigwedge_{k=0}^{m-1} \widehat{c'_k} \wedge \bigwedge_{l=0}^{q-1} \widehat{c_l^{inv'}}) \Leftrightarrow \texttt{false}$$

We write $dp_2 \asymp_{dep} dp_1$ to denote the fact that dp_2 is deadlock-consistent with dp_1. Deadlock consistency expresses the fact that if there is a cycle between two signals x and y in the graph of the source program such that it does not stand for a deadlock, then in the graph of the compiled program, every cycle between x and y must not stand for a deadlock. In the special case where $m = n = p = q = 1$, $x \xrightarrow{\widehat{c'_0}} y$ is deadlock-consistent with $x \xrightarrow{\widehat{c_0}} y$ if $\models (c_0 \wedge c_0^{inv} \Leftrightarrow \texttt{false}) \Rightarrow (c'_0 \wedge c_0^{inv'} \Leftrightarrow \texttt{false})$.

Recall that SDDG(A) and SDDG(C) are two synchronous data-flow dependency graphs, to which we refer respectively as the data dependency representations of the source program and its transformed counterpart produced by the SIGNAL compiler. We assume that they have the same set of nodes, $N_A = N_C$. Let x and y be two signals, we say that C preserves the data dependencies among signals in A if the following conditions are satisfied.

- For every dependency path from the signal x to the signal y in A, there exists a dependency path from x to y in C.
- If there is no deadlock in A, then C introduces no deadlock. In other words, if A is deadlock-free, then it is required that C is deadlock-free.

These conditions can be expressed in terms of the synchronous data-flow dependency graphs as follows.

- For every dependency path from the signal x to the signal y in SDDG(A) at a clock constraint $\widehat{c_1}$, then there exists a dependency path from x to y at a clock constraint $\widehat{c_2}$ in SDDG(C) such that the dependency path in SDDG(C) is effective whenever the dependency path in SDDG(A) is effective.
- If SDDG(A) is deadlock-free, then SDDG(C) is also deadlock-free.

If two SDDGs satisfy the above conditions, we say that SDDG(C) is a *dependency refinement* of SDDG(A) on N_A and `C` is a correct implementation of A. We write SDDG(C) $\sqsubseteq_{dep}$ SDDG(A) to denote the fact that there exists a dependency refinement relation between SDDG(C) and SDDG(A). We formalize the definition of dependency refinement as follows.

Definition 25 (Dependency refinement) *Let* SDDG(A) *and* SDDG(C) *be two synchronous data-flow dependency graphs,* SDDG(C) *is a dependency refinement of* SDDG(A) *if:*

1. *for every dependency path* $dp_1 = \langle x_0, x_1, ..., x_n \rangle$ *in* SDDG(A)*, there exists a dependency path* $dp_2 = \langle x'_0, x_1, ..., x'_m \rangle$ *in* SDDG(C) *such that* $dp_2 \preceq_{dep} dp_1$*,*
2. *for every dependency path* $dp_1 = \langle x_0, x_1, ..., x_n \rangle$ *in* SDDG(A)*, for any dependency path* $dp_2 = \langle x'_0, x_1, ..., x'_m \rangle$ *in* SDDG(C)*, it satisfies* $dp_2 \succeq_{dep} dp_1$*.*

Based on the definitions of reinforcement, deadlock consistency, and dependency refinement relations, important properties are given in Proposition 5.

Proposition 5 *The reinforcement, deadlock consistency and dependency refinement are reflexive and transitive:*

1. $\forall dp, dp \preceq_{dep} dp$
2. $\forall dp, dp \succeq_{dep} dp$
3. $\forall \text{SDDG}(P), \text{SDDG}(P) \sqsubseteq_{dep} \text{SDDG}(P)$
4. *If* $dp_1 \preceq_{dep} dp_2$ *and* $dp_2 \preceq_{dep} dp_3$ *then* $dp_1 \preceq_{dep} dp_3$
5. *If* $dp_1 \succeq_{dep} dp_2$ *and* $dp_2 \succeq_{dep} dp_3$ *then* $dp_1 \succeq_{dep} dp_3$

6. *If* $\text{SDDG}(P_1) \sqsubseteq_{dep} \text{SDDG}(P_2)$ *and* $\text{SDDG}(P_2) \sqsubseteq_{dep} \text{SDDG}(P_3)$ *then* $\text{SDDG}(P_1) \sqsubseteq_{dep} \text{SDDG}(P_3)$

Proof 10 *The proof is based on the definitions of reinforcement, deadlock consistency and dependency refinement.*

Reinforcement *For every dependency path dp, we always have $dp \preceq_{dep} dp$.*
Assume that $dp_1 \preceq_{dep} dp_2$ and $dp_2 \preceq_{dep} dp_3$, we have $(\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j})$ and $(\models \bigwedge_{j=0}^{m-1} \widehat{c'_j} \Rightarrow \bigwedge_{k=0}^{p-1} \widehat{c''_k})$. Thus, $(\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{k=0}^{p-1} \widehat{c''_k})$, or $dp_1 \preceq_{dep} dp_3$.

Deadlock consistency *For every dependency path dp, we always have $dp \asymp_{dep} dp$.*
Because $dp_1 \asymp_{dep} dp_2$ and $dp_2 \asymp_{dep} dp_3$, we have $\models (\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow \texttt{false} \Rightarrow (\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{q-1} \widehat{l'_u}) \Leftrightarrow \texttt{false}$ and $\models (\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{q-1} \widehat{l'_u}) \Leftrightarrow \texttt{false}) \Rightarrow (\bigwedge_{t=0}^{r-1} \widehat{c''_t} \wedge \bigwedge_{z=0}^{s-1} \widehat{l''_z}) \Leftrightarrow \texttt{false}$. Therefore, $\models (\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow \texttt{false} \Rightarrow (\bigwedge_{t=0}^{r-1} \widehat{c''_t} \wedge \bigwedge_{z=0}^{s-1} \widehat{l''_z}) \Leftrightarrow \texttt{false}$, or $dp_1 \asymp_{dep} dp_3$.

Dependency refinement *For every dependency path $dp \in \text{SDDG}(P)$, we have $dp \preceq_{dep} dp$ and $dp \asymp_{dep} dp$, thus $\text{SDDG}(P) \sqsubseteq_{dep} \text{SDDG}(P)$.*
For every dependency path $dp_3 \in \text{SDDG}(P_3)$, there exists a dependency path $dp_2 \in \text{SDDG}(P_2)$ such that $dp_2 \preceq_{dep} dp_3$. Then, there exists a dependency path $dp_1 \in \text{SDDG}(P_1)$ such that $dp_1 \preceq_{dep} dp_2$. Following the transitivity of the reinforcement, we have $dp_1 \preceq_{dep} dp_3$. In the same way, for every dependency path $dp_3 \in \text{SDDG}(P_3)$, any dependency path $dp_1 \in \text{SDDG}(P_1)$ satisfies $dp_1 \asymp_{dep} dp_3$. Therefore, $\text{SDDG}(P_1) \sqsubseteq_{dep} \text{SDDG}(P_3)$.

5.3.2 Adaptation to the SIGNAL compiler

We shall adapt the above definition of dependency refinement to the case of the SIGNAL compiler to prove the preservation of data dependencies among signals when the compiler transforms one SIGNAL program into the intermediate representation form also written in SIGNAL language. We need to consider the following factors as described previously in Section 4.4.3.

A first consideration is that the SIGNAL programs take the inputs from their environment and the register values. Then, they calculate the outputs to react with the environment. In general, the programs can use some local variables to make the output calculations. However, from the outside, the natural observation of the programs is the snapshot of the values of the input and output signals. In our context, it is the snapshot of the dependencies among

the input and output signals. For example, for the program DEC, the observation is the dependencies between the signals (FB,N) at a considered instant.

A second consideration is that in the compilation process of the SIGNAL compiler, the local signals in the source program do not necessarily have counterparts in the compiled program. However, all input and output signals are preserved in the transformations and are represented by identical names in the transformed program. Therefore, it is natural to choose also the observation for the transformed program as the snapshot of the dependencies among the input and output signals.

These considerations let us adapt the above definition of clock refinement as follows. Let SDDG(A) and SDDG(C) be two synchronous data-flow dependency graphs such that they have same set of input and output nodes, $I_A = I_C$ and $O_A = O_C$. We say that C is a correct implementation of A if at any instant, the dependencies among the signals in $I_A \cup O_A$ are also the dependencies among the signals in $I_C \cup O_C$. In other words, SDDG(C) $\sqsubseteq_{dep}$ SDDG(A) on $I_A \cup O_A$.

5.3.3 Proving dependency refinement by SMT

We now discuss a method to check the existence of a refinement between two SDDGs in Definition 25 with the use of a SMT solver. Let SDDG(A) and SDDG(C) be the synchronous data-flow dependency graphs of given input and compiled programs. The set of all common input and output signals between A and C is represented by the common set of input and output nodes in the graphs, $I_A \cup O_A$. For all signals or clock variables in SDDG(C) except the common input and output signals, we added "c" as superscript to distinguish them from the signals in SDDG(A). The variable mapping that maps the non input, output signals from SDDG(A) to the non input, output signals in SDDG(C) is constructed as described in Section 4.4.4. Our aim is proving that SDDG(C) refines SDDG(A) on $I_A \cup O_A$.

To check the existence of the dependency refinement, we traverse the entire graphs SDDG(A) and SDDG(C) to verify the following.

- For every path dp_1 from x to y such that $x, y \in I_A \cup O_A$ in SDDG(A), there exists a reinforcement path dp_2 from x to y in SDDG(C).
- And for every path dp_1 from x to y such that $x, y \in I_A \cup O_A$ in SDDG(A), any path dp_2 from x to y in SDDG(C) is deadlock-consistent with the path dp_1.

Consider two dependency paths $dp_1 = \langle x_0, x_1, ..., x_n \rangle$ and $dp_2 = \langle x'_0, x_1, ..., x'_m \rangle$ in SDDG(A) and SDDG(C), respectively, where $\widehat{c_i}$ and $\widehat{c'_j}$ denote the clock constraints $m_E(x_i, x_{i+1})$ and

$m_E(x'_j, x'_{j+1})$. As in Definition 22, dp_2 is a reinforcement of dp_1 iff the following formula is valid.

$$\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j} \tag{5.1}$$

In the same way, to check the deadlock consistency between dp_1 and dp_2, we have to check the validity of the following formula.

$$(\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{c_j^{inv}}) \Leftrightarrow \texttt{false} \Rightarrow (\bigwedge_{k=0}^{m-1} \widehat{c'_k} \wedge \bigwedge_{l=0}^{q-1} \widehat{c_l^{inv'}}) \Leftrightarrow \texttt{false} \tag{5.2}$$

To solve the validity of the formula above under the variable mapping, a SMT solver is needed since this formula involves non-Boolean variables and *uninterpreted functions* as in our abstraction of clock semantics. In our case, we shall ask the solver to check whether the formula $\neg$(5.1) is unsatisfiable, since this formula is unsatisfiable iff $\models$ (5.1). We do the same for the formula (5.2) meaning that we ask the solver to check whether the formula $\neg$(5.2) is unsatisfiable.

The clock constraints which are constructed from Boolean or numerical variables and uninterpreted functions belong to a part of first-order logic which has a *small model* property according to [28]. This means that satisfiability can be established by examining a certain finite cardinality of models, and it can be solved efficiently.

5.3.4 Implementation

Let A and C be the source and compiled programs. Cp denotes an unverified compiler which compiles the source program A into the compiled program $C = IR(A)$ or a compilation error. We now associate Cp with a validator that checks that the compiled program preserves the data dependencies among signals in A. Accordingly, we get the following derived compiler:

```
1 if (Cp(A) == Error) return Error;
2 else
3 {
4    if (SDDG(IR(A)) ⊑dep SDDG(A)) return IR(A);
5    else return Error;
6 }
```

Figure 5.6 shows the main components of the validator. First, it takes the input program A and the counterpart transformed program C. It constructs the corresponding syn-

chronous data-flow dependency graphs. Then, it establishes the first-order logic formulas as the formula 5.1 and the formula 5.2, and the above variables mapping from the transformed program to the input program. Finally, in the solving phase, it checks the validity of the formulas in the previous step to indicate the dependency refinement.

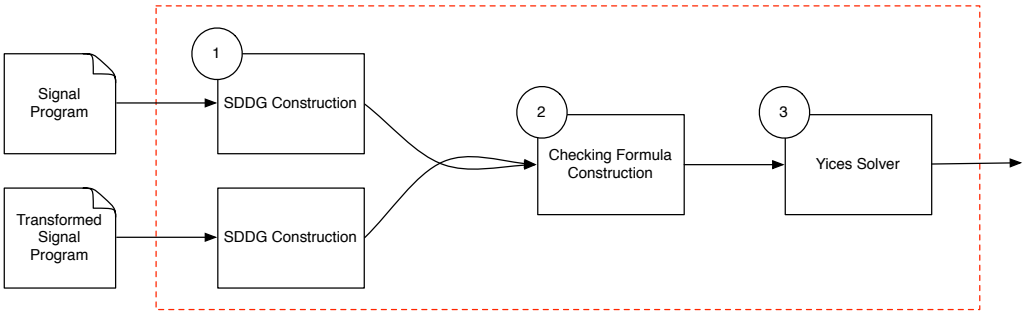

Fig. 5.6 A bird's-eye view of the SDDG translation validation

SDDG construction Given the input program A and its transformed program C, this step will construct the corresponding synchronous data-flow dependency graphs SDDG(A) and SDDG(C) according to the above encoding scheme.

Checking formula construction Consider the graphs SDDG(A) and SDDG(C). This step first establishes the variable mapping. Then, it traverses the graphs to construct the formulas as the formula 5.1 for every pair of paths from x to y in the graphs, and the formula 5.2 for every cycle in the graph SDDG(A). The formulas are the input of the solving phase for checking their validity.

SMT-based proof We delegate checking the validity of the formulas in the previous step to a SMT solver.

Let us illustrate the above process on the program `DEC` and its transformed program `DEC_SEQ_TRA` in Listing 5.1 at the scheduling phase of the SIGNAL compilation process.

Listing 5.1 DEC_SEQ_TRA in Signal

```
1 (| when Tick ^= ZN ^= C_FB ^= C_CLK_12
2  | (| when C_FB ^= FB |)
3  | (| N := (FB when C_FB) default ((ZN - 1) when C_CLK_12)
4     | ZN := N$1 init 1
5     | C_FB := ZN <= 1
```

```
6       | C_CLK_12 := not (ZN <= 1)
7       |)
8   |)
```

In the first step, we shall construct the synchronous data-flow dependency graphs which are depicted in Figure 5.5 and Figure 5.7. Two fresh variables $FB1$ and $ZN3$ are used to replace the expressions FB `when` C_FB and $ZN2$ `when` C_CLK_12. We omit the dependencies among the signals $FB, \widehat{FB}, C\_FB^c$ and $\widehat{C\_FB^c}$ in the graph of the program `DEC_SEQ_TRA`.

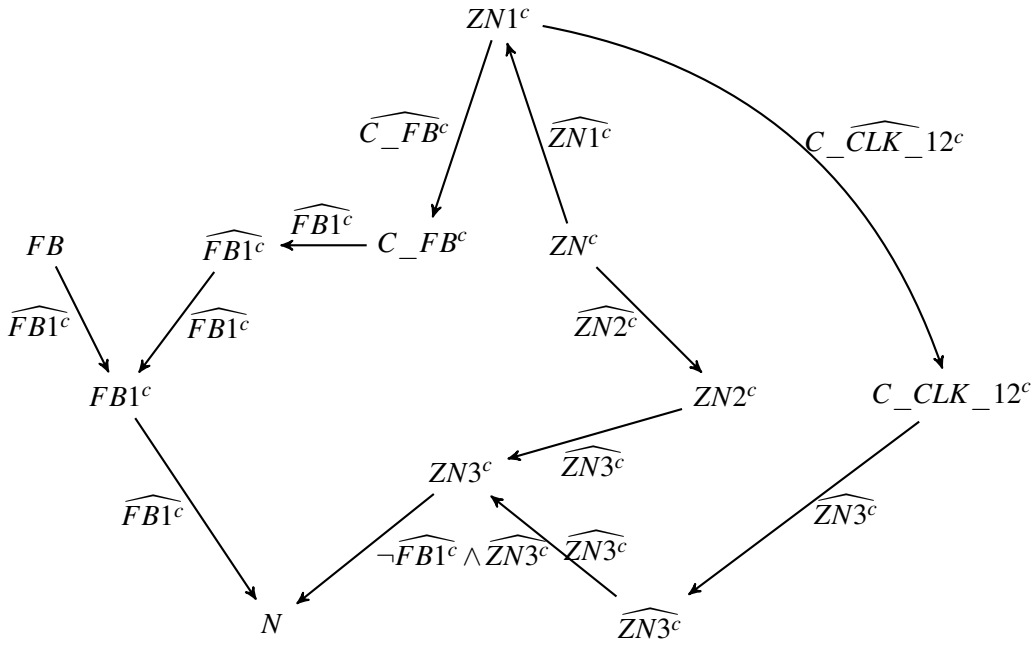

Fig. 5.7 The SDDG of DEC_SEQ_TRA

In the second step, checking formula construction, our tool will establish the variable mapping $\widehat{X_{\texttt{DEC}}} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_{\texttt{DEC\_BASIC\_TRA}}} \setminus \widehat{X_{IO}})$ as in Section 4.4.5.1. Next, it finds all dependency paths from the signal FB to the signal N, and the cycles in the graphs SDDG(`DEC`) and SDDG(`DEC_SEQ_TRA`) to generate the formulas for checking the dependency refinement. In this example, we get the following formula:

$$\widehat{FB} \Rightarrow (\widehat{FB1^c} \wedge \widehat{FB1^c})$$

In the third step, we delegate the checking validity of the formula to YICES solver. We get `unsat` when checking the satisfiability of $\neg(\widehat{FB} \Rightarrow (\widehat{FB1^c} \wedge \widehat{FB1^c}))$, which means the formula is valid. Therefore, we can conclude that SDDG(`DEC_SEQ_TRA`) $\sqsubseteq_{dep}$ SDDG(`DEC`)

on the signals FB and N.

5.4 Precise deadlock detection for SIGNAL compiler

The SIGNAL compiler constructs the data dependencies among signals in a program, represented as a labeled directed graph, in which the labels are polynomials in $\mathbb{Z}/_3\mathbb{Z}$. For each dependency cycle in the graph, it checks the product of the cycle labels. If this product, which is represented as a polynomial, is equal to the *null clock*, then it does not stand for a deadlock. However, consider the *sampling* operator $y := x$ `when` b, in which the clock of the signal y is defined by the *condition clock* $[b]$, which defines the instants where b is present and has value `true`. If the Boolean expression b is a non Boolean relation (e.g., a comparison between numerical expressions), then the $\mathbb{Z}/_3\mathbb{Z}$ abstraction considers the expression's clock instead. This yields an approximation of the actual dependency which may cause the compiler to approximate dependencies, like in the example above.

In this section, we propose a more precise deadlock detection approach. Our approach permits the compiler to detect deadlocks with numerical expressions. The data dependencies among signals are represented by a synchronous data-flow dependency graph-like, called SDDG$^+$. A SDDG$^+$ is a labeled directed graph in which each node is a signal or clock variable and each edge represents a dependency between two nodes. Each edge is labeled by a condition at which the dependency is effective. We use the Boolean-interval abstraction that was originally proposed by Gamatié et al. [57] to encode the clock labels. That means every signal is associated with a pair of the form *(clock, value)*, where *clock* is a Boolean function and *value* is a Boolean or numerical function, abstracted as an interval. We use a SMT solver to reason on the labels when deciding a dependency cycle in a SDDG$^+$ to stand for a deadlock. We show how our approach addresses the limitation of the current deadlock detection used in the SIGNAL compiler through a concrete example.

5.4.1 Deadlock detection in the SIGNAL compiler

Before generating the executable code on a given architecture, the compilation performed by the SIGNAL compiler aims at proving the *reactivity* and the *determinism* of programs by modeling the synchronization relations and checking the absence of cyclic data dependencies in program specifications. In our consideration, the compiler needs to answer the following question: *"Is the program deadlock-free?"*. To answer this question, the Signal compiler uses the *Graph of Conditional Dependencies* (GCD) in which a dependency be-

tween data is conditioned by a clock. We will illustrate this technique with an example (a more detailed discussion is presented in [97]).

The SIGNAL program in Listing 5.2 emits an integer sequence v whose ith value v_i is the double of the input x_i depending on the other input c.

Listing 5.2 CycleDependency in SIGNAL

```
process CycleDependency=
(? integer x, c;
 ! integer v)
 (| y := (v when (c <= 0)) default x
  | u := y + x
  | v := u when (c >= 1)
  |)
where integer y, u
end;
```

A possible run of the program is depicted by the following trace:

```
x   1  3  2  2  #  #  4  7  9 ...
c   1  3  0  1 -1 -2  3  6  2 ...
y   1  3  2  2  #  #  4  7  9 ...
u   2  6  4  4  #  #  8 14 18 ...
v   2  6  #  4  #  #  8 14 18 ...
```

Apply the $\mathbb{Z}/3\mathbb{Z}$ encoding scheme on the program `CycleDependency`, we have the following algebraic coding, where we introduce three fresh variables c_1, c_2 and v_1 to replace the expressions $(c <= 0), (c >= 1)$ and $(v \texttt{ when } c_1)$, respectively:

$$\begin{aligned} y^2 &= v_1^2 + (1 - v_1^2)x^2 \\ v_1^2 &= v^2(-c_1 - c_1^2) \\ u^2 &= y^2 = x^2 \\ c_1^2 &= c^2 = c_2^2 \\ v^2 &= u^2(-c_2 - c_2^2) \end{aligned}$$

Given the algebraic encoding above, the SIGNAL compiler can establish the dependencies among signals y, u and v as a graph of conditional dependencies which is depicted in Figure 5.8.

Based on GCD graphs, the SIGNAL compiler identifies the potential deadlocks in the program. Such a bad dependency between signals will appear as a cycle in the graph if the product of the labels of its edges is not the null clock.

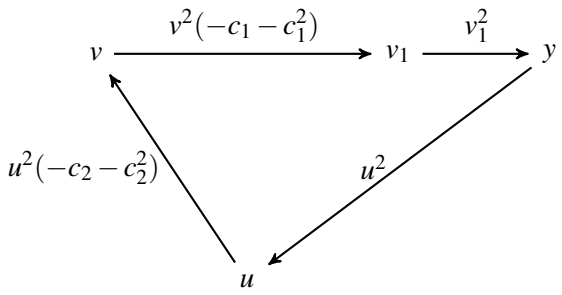

Fig. 5.8 Dependencies among y, u, v

However, for the *sampling* operator $y := x$ `when` b, if the expression b is a non Boolean relation, then the $\mathbb{Z}/_3\mathbb{Z}$ encoding considers this condition clock as an indeterminate value. This naturally yields over-approximated detection when dealing with non Booleans values.

The `CycleDependency` program and its dependencies among the signals y, u and v in Figure 5.8 exemplify the over-approximated detection (we omit the dependencies among v, v_1, y and u). These dependencies introduce a cycle in the graph. To verify that this cycle is not a deadlock, the SIGNAL compiler calculates the following product of the labels is null clock:

$$v^2(-c_1-c_1^2) * v_1^2 * u^2 * u^2(-c_2-c_2^2)$$

Replacing the definitions of v^2, v_1^2 and u^2, we have:

$$x^2(-c_2-c_2^2)(-c_1-c_1^2) * x^2(-c_2-c_2^2)(-c_1-c_1^2) * x^2 * x^2(-c_2-c_2^2)$$

With the current clock calculus, the compiler concludes that this cycle may cause a deadlock since at some instants the above product is different from 0 (e.g. when the signals x, c_1, c_2 are present and both c_1 and c_2 hold the value `true`, or $x^2 = 1, c_1 = 1$, and $c_2 = 1$). But in fact, c_1 and c_2 cannot hold the value `true` at the same time.

The above limitation makes the SIGNAL compiler reject some possibly valid programs. To address this issue, we propose a new deadlock detection to better handle numerical operators.

5.4.2 A more precise deadlock detection

In this section, we will present a more precise deadlock detection technique compared to the technique currently used by the SIGNAL compiler. This technique uses the concept of

SDDG$^+$ to express the data dependencies among the signals in a program.

In presenting our deadlock detection below, we first describe the abstraction which is used to encode the clock relations of a SIGNAL program and the construction of a SDDG$^+$.

5.4.2.1 A Boolean-interval abstraction for clock semantics

Let $X = \{x_1, ..., x_n\}$ be the set of all signals in program P. With each signal x_i, we attach a Boolean variable $\widehat{x_i}$ to encode its clock and a variable $\widetilde{x_i}$ of same type as x_i to encode its value. Formally, the abstract values which represent the semantics of the program can be computed using the following functions:

$\hat{}: X \longrightarrow \mathbb{B}$ associates a signal with a Boolean value
$\tilde{}: X \longrightarrow \mathbb{D}$ associates a signal with a value of the same type

The composition of SIGNAL processes corresponds to logical conjunctions. Thus the abstract model of P will be a conjunction $\Phi(P) = \bigwedge_{i=1}^{n} \phi(eq_i)$ whose atoms are $\widehat{x_i}, \widetilde{x_i}$, where $\phi(eq_i)$ is the abstraction of statement eq_i (statement using the Signal primitive operators), and n is the number of statements in the program. In the following, we present the abstraction corresponding to each SIGNAL operator. There are two definitions of Φ according to the type of the signal on the left hand side in each equation: (1) stands for numerical type and (2) is for logical type. The Boolean-interval abstraction preserves the behaviors of the program being abstracted, in other word, it is sound. The details of the abstraction and the soundness proof are presented in [57].

Stepwise functions The functions which apply on signal values in the primitive *stepwise functions* are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`), and numerical operators (`+`, `-`, `*`, `/`) (denoted by $\square$). In our implementation, we replace the operation results by intervals and their representation in logic context by *uninterpreted functions*.

The abstraction $\phi(y := f(x_1, ..., x_n))$ of stepwise functions is defined as follows:

$$\bigwedge_{i=1}^{n}(\hat{y} \Leftrightarrow \hat{x_i}) \wedge (\hat{y} \Rightarrow \tilde{y} \in \phi(f(x_1, ..., x_n))) \quad (1)$$
$$\bigwedge_{i=1}^{n}(\hat{y} \Leftrightarrow \hat{x_i}) \wedge (\hat{y} \Rightarrow \tilde{y} \Leftrightarrow \phi(f(x_1, ..., x_n))) \quad (2)$$

where the abstraction $\phi(f(x_1, ..., x_n)$ is defined by induction as follows:

- $\phi(\texttt{true}) = \texttt{true}$ and $\phi(\texttt{false}) = \texttt{false}$.

- $\phi(x) = \tilde{x}$ if x is of Boolean type; $\phi(x) =$ `true` if x is of event type.
- $\phi(x) = inv(x)$ if x is of non-Boolean type, where $inv(x)$ is the interval of x.
- $\phi(x_1 \; and \; x_2) = \widetilde{x_1} \wedge \widetilde{x_2}$.
- $\phi(x_1 \; or \; x_2) = \widetilde{x_1} \vee \widetilde{x_2}$.
- $\phi(not \; x_1) = \neg\widetilde{x_1}$.
- $\phi(x <= c) = \tilde{x} \in (-\infty, c]$; $\phi(x < c) = \tilde{x} \in (-\infty, c)$.
- $\phi(x_1 <= x_2) = (\tilde{x} \in \phi(x_1 - x_2)) \wedge \tilde{x} \in (-\infty, 0]$, where x is a fresh variable.
- $\phi(x_1 \,\square\, x_2) = \widetilde{\square}(\phi(x_1), \phi(x_2))$, an approximation of numerical operations on intervals, corresponding to $\square$ as in [5]. For example, for addition operation $i \tilde{+} j \equiv [i^- + j^-, i^+ + j^+]$, where i^- and i^+ are respectively the lower and upper bounds of the interval i.

Delay Considering the *delay* operator, $y := x\$1$ `init` a, its encoding $\phi(y := x\$1$ `init` $a)$ contributes to $\Phi(P)$ with the following conjunct. This encoding requires that at any instant, signals x and y have the same clock. If they are numerical signals, then they have the same interval. Otherwise, we introduce a memorization variable $m.x$ that stores the last value of x. The next value of $m.x$ is $m.x'$ and it is initialized to a in $m.x_0$.

$$(\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \in \phi(x) \vee \tilde{y} = a)) \quad (1)$$
$$(\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \widetilde{m.x} \wedge \widetilde{m.x'} \Leftrightarrow \tilde{x})) \wedge (\widetilde{m.x_0} \Leftrightarrow a) \quad (2)$$

Merge The encoding of the *merge* operator, $y := x$ `default` z, contributes to $\Phi(P)$ with the following conjunct:

$$(\hat{y} \Leftrightarrow \hat{x} \vee \hat{z}) \wedge (\hat{y} \Rightarrow (\hat{x} \wedge (\tilde{y} = \tilde{x})) \vee (\neg\hat{x} \wedge (\tilde{y} = \tilde{z})))$$

Sampling The encoding of the *sampling* operator, $y := x$ `when` b, contributes to $\Phi(P)$ with the following conjunct:

$$(\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \tilde{b})) \wedge (\hat{y} \Rightarrow (\tilde{y} = \tilde{x}))$$

Composition Consider the composition of two processes P_1 and P_2. Its abstraction $\phi(P_1|P_2)$ is defined by $\phi(P_1) \wedge \phi(P_2)$.

Clock relations Given the above rules, we can obtain the following abstraction for derived operators on clocks. Here, z is a signal of type *event*:

- $\phi(z := \hat{x}) = (\hat{z} \Leftrightarrow \hat{x}) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(x \hat{} = \ y) = \hat{x} \Leftrightarrow \hat{y}$
- $\phi(z := x \hat{} + \ y) = (\hat{z} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(z := x \hat{} * \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(z := x \hat{} - \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$
- $\phi(z := \ \texttt{when}\ b) = (\hat{z} \Leftrightarrow (\hat{b} \wedge \tilde{b})) \wedge (\hat{z} \Rightarrow \tilde{z})$

Example Assume that the intervals for the signals are given as $x \in (-\infty, +\infty)$, $c \in (-\infty, +\infty)$, $y \in (-\infty, +\infty)$, $u \in (-\infty, +\infty)$, $v \in (-\infty, +\infty)$. Applying the abstraction rules above, the abstraction of the program `CycleDependency` is represented by the following first-order logic formula, where we introduce three fresh variables c_1, c_2 and v_1 to replace the expressions $(c <= 0), (c >= 1)$ and $(v\ \texttt{when}\ c_1)$, respectively:

$$
\begin{aligned}
&(\hat{y} \Leftrightarrow \widehat{v_1} \vee \hat{x}) \wedge (\tilde{y} \in (-\infty, +\infty)) \wedge (\widehat{v_1} \Leftrightarrow \hat{v} \wedge \widehat{c_1} \wedge \widetilde{c_1}) \wedge (\widetilde{v_1} \in (-\infty, +\infty)) \\
\wedge &(\widehat{c_1} \Leftrightarrow \hat{c}) \wedge (\widetilde{c_1} \Leftrightarrow (\tilde{c} \in (-\infty, 0])) \wedge (\hat{u} \Leftrightarrow \hat{y} \Leftrightarrow \hat{x}) \wedge (\tilde{u} \in (-\infty, +\infty)) \\
\wedge &(\hat{v} \Leftrightarrow \hat{u} \wedge \widehat{c_2} \wedge \widetilde{c_2}) \wedge (\tilde{v} \in (-\infty, +\infty)) \wedge (\widehat{c_2} \Leftrightarrow \hat{c}) \wedge (\widetilde{c_2} \Leftrightarrow (\tilde{c} \in [1, +\infty)))
\end{aligned}
$$

A SDDG$^+$ for a given program is a labeled directed graph in which each node is a signal or clock variable and each edge represents the dependency between nodes. Each edge is labeled by a first-order logic formula which represents the clock at which the dependency between the extremity nodes is effective. Formally, a SDDG$^+$ is defined as follows:

Definition 26 (SDDG$^+$) *A* SDDG$^+$ *associated with a process* P *is a tuple* $G = \langle N, E, I, O, C, m_N, m_E \rangle$ *where:*

- *N is a finite set of nodes, each of which represents the equation defining a signal or a clock.*
- *$E \subseteq N \times N$ is the set of dependencies between nodes.*
- *$I \subseteq N$ is the set of input nodes.*
- *$O \subseteq N$ is the set of output nodes.*

- *C is the set of first-order logic formulas over a set of clocks in the Boolean-interval abstraction.*
- *$m_N : N \longrightarrow C$ is a mapping labeling each node with a clock; it defines the existence condition of a node.*
- *$m_E : E \longrightarrow C$ is a mapping labeling each edge with a clock constraint; it defines the existence condition of an edge.*

The clock labeling in SDDG$^+$ provides a dynamic dependency feature. This clock imposes the following property: *An edge cannot exist if one of its two extremity nodes does not exist.* This property can be translated in our Boolean-interval abstraction as:

$$\forall (x,y) \in E, m_E(x,y) \Leftarrow (m_N(x) \wedge m_N(y))$$

We denote the fact that there exists a dependency between two nodes x and y at a clock condition $m_E(x,y) = \hat{c}$ by $x \xrightarrow{\hat{c}} y$. A *dependency path* from x to y is any set of nodes $s = \{x_0, x_1, ..., x_k\}$ such that (an edge is a special case when $k = 1$):

$$x = x_0 \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{k-1}}} x_k = y$$

The dependencies among signals for the primitive operators of the SIGNAL language are the same as the encoding in Table 5.2 except that the clock constraints which label the edges in the graph are represented in the Boolean-interval abstraction.

Following the above construction rules, we can obtain the SDDG$^+$ in Figure 5.9, for the simple program `CycleDependency` (we omit the parts of the graph that represent the dependencies of c_1 and c_2 on c). In this graph, the clock labels are defined in the above Boolean-interval abstraction.

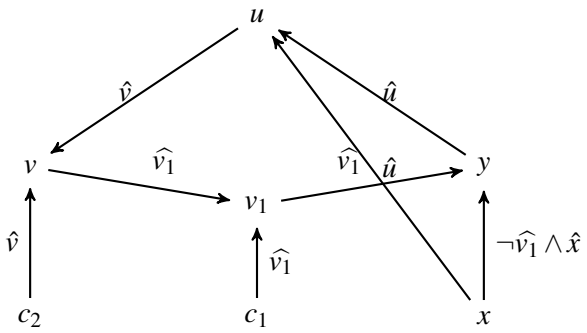

Fig. 5.9 The SDDG$^+$ of CycleDependency

5.4.3 Precise deadlock detection

In this section, we shall present a more precise deadlock detection technique which is based on the concept of SDDG$^+$ along with the Boolean-interval abstraction. We shall show the performance comparison of our technique with the current technique of the SIGNAL compiler using a concrete example.

5.4.3.1 Deadlock definition

Let G be a SDDG$^+$, and $x \xrightarrow{\hat{c}} y$ be an edge in G. Assume that there exists a path from y to x, that makes a dependency cycle between x and y. A cycle of dependencies, standing for a deadlock, is defined with a similar meaning to the one in GCD. We say that such cycle is a deadlock in G iff the dependencies of x to y and vice-versa are effective at the same time. Transposing the notion of deadlock to SDDG$^+$ graphs, we have the following definition:

Definition 27 (Deadlock) *Let $G = \langle N, E, I, O, C, m_N, m_E \rangle$ be a* SDDG$^+$*, and $p_c = x_1, ..., x_n, x_1$ be a cycle in G. p_c is said a deadlock in G if there exists some interpretations that make the first-order logic formula $m_E(x_1, x_2) \wedge m_E(x_2, x_3) \wedge ... \wedge m_E(x_n, x_1)$* `true`.

Based on the definition of deadlock in SDDG$^+$ graphs, we can have the definition of *deadlock-free* of a SDDG$^+$ as follows:

Definition 28 (Deadlock-free) *Let $G = \langle N, E, I, O, C, m_N, m_E \rangle$ be a* SDDG$^+$*. G is deadlock-free iff for every cycle $(x_1, ..., x_n, x_1)$ in G, the conjunction of all the conditions associated with the cycle $m_E(x_1, x_2) \wedge m_E(x_2, x_3) \wedge ... \wedge m_E(x_n, x_1) \Leftrightarrow$* `false` *is identically* `false`.

Obviously, the fact that the conjunction of first-order logic formulas associated with the cycle, in which the dependencies are effective, is `false` indicates that a deadlock does not exist if all the dependencies of the cycle in the SDDG$^+$ graph cannot be present at the same time.

5.4.3.2 Proving deadlock free by SMT

Given the SDDG$^+$ of a program, we introduce an approach to check that the graph is deadlock-free. It is implemented with a SMT solver [51, 131]. A SMT solver decides the satisfiability of arbitrary logic formulas of linear real and integer arithmetic, scalar types, other user-defined data structures, and uninterpreted functions. If the formula belongs to the decidable theory, the solver gives two types of answers: `sat` when the formula has a model (there exists an interpretation that satisfies it); or `unsat` otherwise. In our case, the formulas

which label the edges of the graph are over Boolean variables and uninterpreted functions, thus the solving is decidable and very efficient [3, 21].

Following Definition 28, we shall traverse the entire graph to find all cycles such as $(x_1,...,x_n,x_1)$ and for each of them, we verify the following property. Notice that here, we do not provide any specific algorithm to find the cycles in a directed graph, interested readers can be refer to any research on this problem (e.g. the work of Johnson [82]).

$$m_E(x_1,x_2) \wedge m_E(x_2,x_3) \wedge ... \wedge m_E(x_n,x_1) \Leftrightarrow \texttt{false}$$

It means that the basic element we have to prove is that given a dependency cycle and the conjunction of its labels, this conjunction formula is always evaluated to the value `false`.

Consider a dependency cycle $x_1 \xrightarrow{\widehat{c_1}} x_2 \xrightarrow{\widehat{c_2}} ... \xrightarrow{\widehat{c_{n-1}}} x_n \xrightarrow{\widehat{c_n}} x_1$. This cycle does not stand for a deadlock iff the formula $(\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow false)$ is valid within the logical context defined by the abstraction of `P`. The checking of this condition can be implemented by asking a SMT solver to check $\models (\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow \texttt{false})$. In the equal way, we can ask the SMT solver to check the formula $\neg(\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow \texttt{false})$ is unsatisfiable, or $M \not\models \neg(\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow \texttt{false})$.

5.4.3.3 Implementation and illustrative example

We describe the main steps of our approach, and the techniques we use to implement them. This implementation can be integrated in the existing Polychrony toolset [77] to check that a SDDG$^+$ graph is deadlock-free.

At a high level, our tool, which is depicted in Figure 5.10, works as follows. First, it takes the input program `P`, and constructs the corresponding SDDG$^+$ graph. It finds all cycles in the graph. Finally, in the solving phase, it checks the validity of the formula $(\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow \texttt{false})$ for each cycle $(x_1,...,x_n,x_1)$.

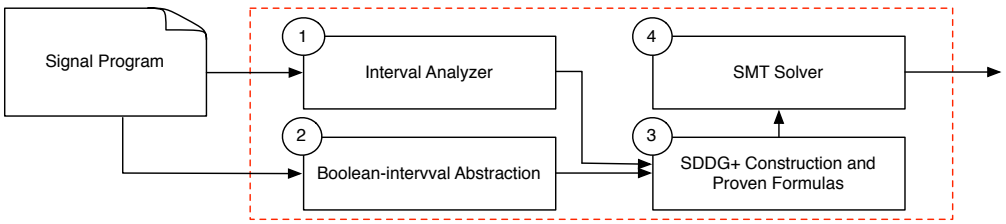

Fig. 5.10 An overview of our approach

Interval analyzer This step determines the interval of every signal in the program. For every input signal, it is assumed that its interval is known. This step applies the algorithm presented in [68]. The tool in [88] is used to compute an over-approximation of the variation interval of each numerical signal.

Abstraction The input program is encoded according to the Boolean-interval abstraction scheme in Section 5.4.2.1. The output of this step is a first-order logic formula.

SDDG$^+$ construction After obtaining the interval analysis and the abstract model of the input program, this step will construct the corresponding SDDG$^+$ graph according to the rules in Table 5.2. The labels of edges in the graph are encoded as first-order logic formulas based on the above Boolean-interval abstraction scheme. In this step, the tool also detects all dependency cycles in the graph and produces the conjunction formula of all clock labels for each cycle as input for the solver.

SMT-based proof We delegate checking the validity of the formulas to a SMT solver. Our implementation uses the Yices [51] solver.

Let us illustrate the above deadlock detection technique on the program `CycleDependency`. In the first step, we shall determine the variation interval for all signals in the program. We assume that the variation intervals for input signals x and c are $(-\infty,+\infty),(-\infty,+\infty)$. After the analysis, we get: $x \in (-\infty,+\infty)$, $c \in (-\infty,+\infty)$, $y \in (-\infty,+\infty)$, $u \in (-\infty,+\infty)$, $v \in (-\infty,+\infty)$. Recall that we rewrite equation at lines (4) and (6) as follows:

```
1 (| y := v1 default x
2  | v1 := v when c1
3  | c1 := c <= 0
4  | v := u when c2
5  | c2 := c >= 1
6  |)
```

In the second step, Boolean-interval abstraction, our tool will translate the program into a logic formula Φ according to the above abstract scheme. Let us focus on the clocks of signals y,u, and v, which are given as follows:

$$\hat{y} \Leftrightarrow \widehat{v_1} \vee \hat{x}, \widehat{v_1} \Leftrightarrow \hat{v} \wedge \widehat{c_1} \wedge (\tilde{c} \in (-\infty,0])$$
$$\hat{v} \Leftrightarrow \hat{u} \wedge \widehat{c_2} \wedge (\tilde{c} \in [1,+\infty)), \hat{u} \Leftrightarrow \hat{y} \Leftrightarrow \hat{x}, \widehat{c_1} \Leftrightarrow \widehat{c_2} \Leftrightarrow \hat{c}$$

In the third step, we construct the SDDG$^+$ graph in Figure 5.9. Here, we detect that there

exists a dependency cycle (v, v_1, y, u, v) in the graph, the tool then generates the formula, referred to as φ, to delegate to the SMT solver as follows:

$$(\widehat{v_1} \wedge \widehat{v_1} \wedge \hat{u} \wedge \hat{v}) \Leftrightarrow \texttt{false}$$

In the logical context defined by Φ, replacing the definitions of $\widehat{c_1}, \widehat{c_2}, \hat{u}, \hat{v}$ and $\widehat{v_1}$ in the program abstraction model, we get:

$$\begin{aligned}&(\hat{x} \wedge \hat{c} \wedge (\tilde{c} \in [1, +\infty)) \wedge \hat{c} \wedge (\tilde{c} \in (-\infty, 0]) \wedge \hat{x} \wedge \hat{c} \wedge (\tilde{c} \in [1, +\infty))\\ &\wedge \hat{c} \wedge (\tilde{c} \in (-\infty, 0]) \wedge \hat{x} \wedge \hat{x} \wedge \hat{c} \wedge (\tilde{c} \in [1, +\infty))) \Leftrightarrow \texttt{false}\end{aligned}$$

With the Yices solver, we will get `unsat` when checking the satisfiability of $\neg\varphi$, which means that φ is valid. Thus, the graph is deadlock-free.

Our deadlock detection technique is more precise than the current technique used by the SIGNAL compiler when dealing with numerical expressions. It admits less erroneous (or "spurious") decision on deadlock detection than the current technique. That means that when our approach is applied on the SIGNAL compiler, it will make the compiler avoid rejecting valid programs. The reason why our technique is more expressive than the current one is that it uses a more suitable and precise abstraction for numerical expressions. For instance, here, our tool can detect that the two signals v and v_1 cannot be present at the same time.

5.5 Discussion

Much work has been performed before in the area of dependency-based program representations. Dennis opened up the area of *data flow computation* in [47]. The *data flow graph* [113, 114] represents global data dependency at the operator level, called the *atomic* level in [87]. Transformations that involve both control and data dependency cannot be specified in a consistent manner with this form since control is represented by a conventional control flow graph. The data dependencies among variables are presented statically in the graph.

The GCD [13, 97] is used to check deadlocks in a SIGNAL program and schedule instructions. The edges of a GCD indicate the data dependencies among signals and clock variables, they are labeled by clocks. These clocks are polynomials whose coefficients range over the finite field $\mathbb{Z}/_3\mathbb{Z}$. These clocks represent the conditions at which the dependencies are valid, meaning that the dependencies can be represented dynamically. Based on

the concept of GCD, we have defined synchronous data-flow dependency graph in which the dependencies among signals can be represented dynamically by conditioning the dependency between two signals with a Boolean expression. They differ from GCD in the labels of the edges in the graphs. In synchronous data-flow graphs, the labels are clocks which are encoded as first-order logic formulas. The synchronous data-flow dependency graphs are used as the common semantics framework of our translation validation for the *scheduling* phase of the SIGNAL compiler. We formalize the notion of "correct transformation" as a refinement relation between two synchronous data-flow dependency graphs. This relation expresses the preservation of the dependencies, whose existence can be checked by using a SMT solver. We also propose a technique to implement and integrate our validator within the POLYCHRONY toolset by the use of YICES solver.

In addition, we also propose a more precise deadlock detection approach for deadlock-free checking of synchronous programs written in SIGNAL language. Our approach permits the compiler avoiding emitting spurious decision on deadlock detection while the current technique does when dealing with numerical expressions. In our solution, the data dependencies among signals are represented by SDDG graphs, in which the nodes are signals or clock variables, edges are dependencies. Each edge is labeled by a condition expressed as a first-order logic formula at which the dependency is effective. We use a SMT solver to reason on the labels when deciding a dependency cycle in a SDDG to stand for a deadlock. In our work, we use the interval-Boolean abstraction as a more suitable abstraction to represent the condition of dependencies among signals to make a more efficient deadlock detection technique. The concept of interval-Boolean abstraction was introduced in [57] by Gamatié et al. to make a more efficient static analysis of SIGNAL compiler.

The deadlock detection analysis presented in this chapter to the project POP [112], as such, it complements the case study of [103] on synthesis safety wrappers for polychronous specifications through polyhedral analysis, and generalises the work presented in [83] on false-loop detection using SMT-solving to the case of a polyhedral analysis.

CHAPTER 6

EVALUATING SDVG TRANSLATION VALIDATION: FROM SIGNAL TO C

In Chapter 4 and Chapter 5, we provide approaches that are based on translation validation technique to prove the preservation of clock semantics and data dependency in the compilation process of the SIGNAL compiler. In this chapter, we describe how the preservation of value-equivalence of variables can be proved based on translation validation of synchronous data-flow value-graphs. It focuses on proving that every output variable in the original program and its counterpart in the transformed program, the generated C code, have the same values. The computation of all output variables and their counterparts is represented by a formal representation, called *Synchronous Data-flow Value-Graph* (SDVG).

This graph symbolically represents the computation of the outputs in the original program and their counterparts in the transformed program. The same structures are shared in the graph, meaning that they are represented by the same nodes. Assume that we want to show that two output variables have the same value. We simply need to check that they are represented by graphs which are rooted at the same graph node. We manage to realize the check by *normalizing* SDVGs using some rewrite rules.

For a transformation, our validator takes the input program and its transformed counterpart and constructs the corresponding SDVG for each output variable. Then it checks that for every output variable in the input program and its counterpart in the transformed program, they have the same values. If the result says that there exists any non-equivalence then the compiler emits a compilation error. Otherwise, the compiler continues its work. The integration of this verification process into the compilation process can be depicted as in Figure 6.1. The remainder of this chapter is organized as follows. Section 6.1 describes how code can be generated, as the final step of the compilation process, following different schemes. Section 6.2 illustrates the concept of SDVG and the verification procedure. In

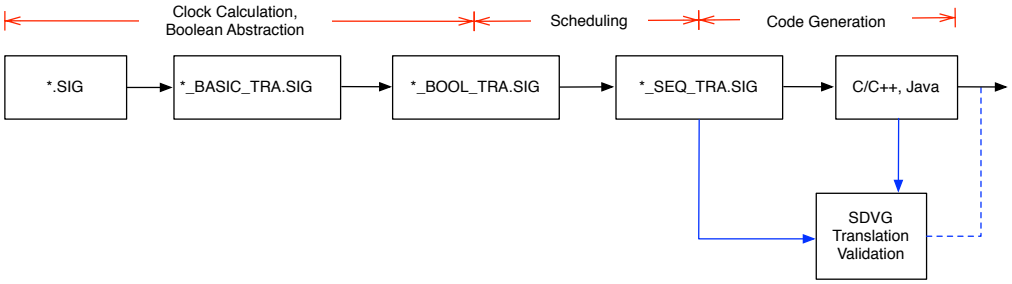

Fig. 6.1 A bird's-eye view of the verification process

Section 6.3, we consider the formal definition of SDVG and the representation of SIGNAL program and generated C code as SDVGs. Section 6.4 addresses the mechanism of the verification process based on the rewrite rules and the normalization of a SDVG. It also presents the application of the verification process to the SIGNAL compiler, and its integration in the Polychrony toolset [77]. Section 6.5 discusses some related works, concludes our work and outlines future directions.

6.1 Code generation in SIGNAL compiler

Code generation is the final step in the compilation process of SIGNAL compiler as depicted in Figure 6.2. When a program P has no deadlocks and is free of clock constraints one can generate code for P with the code generation functionalities of the POLYCHRONY toolset. The code can be generated for different general purpose languages (C, C^{++}, and JAVA) on different architectures. The generated code in this case is called *reactive code*. However, one can generate a *defensive code* when the program has clock constraints; in this mode, alarms are emitted when a contraint is violated during the simulation.

6.1.1 The principle

The principle of code generation [19] is based on the use of the clock hierarchy and the graph of conditional dependencies. The code generation follows the general scheme that is depicted in Figure 6.2. The generated code contains a main program which controls the *step block*. The step block consists of a step scheduler that drives the execution of its step component and updates the state variables corresponding to *delay* operators and local variables. The execution of the step block is scheduled by the *step scheduler*. The step component can be hierarchical, which consists of a set of sub-components called *clusters*.

The step component has its own local step scheduler. The step block communicates with its environment through the IO container.

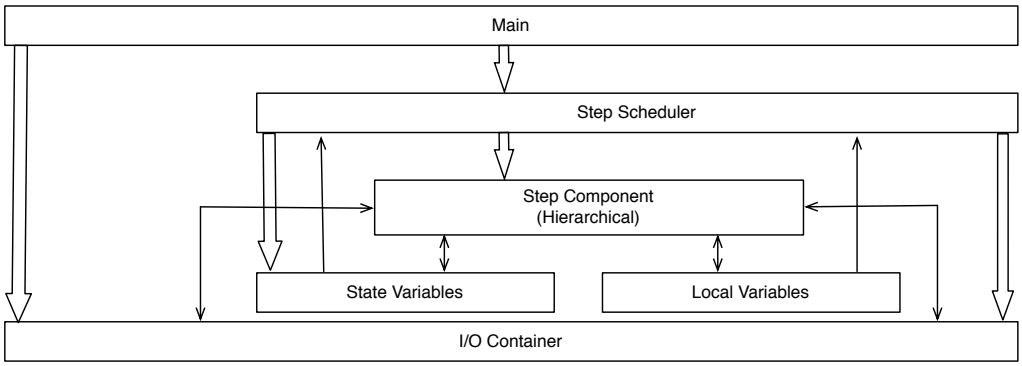

Fig. 6.2 Code generation: General scheme

In general, the generated program consists of several files. We consider that the target language code is C. For a process *P*, a main program is defined in the file *P_main.c*, the program which contains the step block called body program is defined in the file *P_body.c*. An input-output program which contains the IO container is defined in the file *P_io.c*. Each component of the generated code can be seen as a SIGNAL process. Then they can be reused in an embedding SIGNAL process.

6.1.1.1 The main program

The pseudo-code in Listing 6.1 shows the structure of the main program. It first opens the IO communication channels with the program environment and calls the initialization function. If everything goes fine then it calls the step function repeatedly in an infinite loop to interact with the environment. The infinite loop can be stopped if the step function returns the error code 0, meaning that the input stream is empty. And the main program will close the communication channels. All the called functions are defined in the body program.

Listing 6.1 Structure of P_main.c

```
1 EXTERN int main()
2 {
3    logical code;
4    P_OpenIO();
5    code = P_initialize();
6    while(code)
7    {
```

```
8         P_stepIO_begin();
9         code = P_step();
10        P_stepIO_end();
11    }
12    P_CloseIO();
13 }
```

6.1.1.2 The step block

Once the IO communication channels and the initialization are completed, the step function *P_step*() is responsible for the calculation of the effect of one synchronous step of the system to interact with the environment. It is the essential part of the concrete code. It reads data from the input streams, computes the outputs and writes the results to the output streams. In the POLYCHRONY toolset, the implementation of the step function can be done in many schemes based on the clock hierarchy and the graph of conditional dependencies. These code generation schemes consist of:

- Global code generation: sequential code, clustered code with static scheduling, clustered code with dynamic scheduling.
- Modular code generation.
- Distributed code generation.

The next section will describe the sequential, inlining code generation of the step block. For other code generation schemes, interested readers can refer to [10, 64, 97].

6.1.1.3 The IO container

The IO container implements the communication of the generated program with the environment in case the being compiled process contains input and output signals. In the simulation mode, each input or output signal communicates with the environment via a file as the input stream or output stream. The IO container in Listing 6.2 consists of global functions for opening, closing all files, and for reading and writing data for each input and output signal.

Listing 6.2 Structure of P_io.c

```
1 EXTERN void P_OpenIO()      EXTERN void P_CloseIO()
2 {                           {
3    fra = fopen(...);           ...
4    if (!fra) {                 fclose(fwx);
```

```
5       ...                     }
6       exit(1);
7    }                          EXTERN int r_P_a(integer *a)
8    fwx = fopen(...);          {
9    if (!fwx) {                   return (fscanf(fra,"%d",a) != EOF);
10      ...                     }
11      exit(1);
12   }                          EXTERN void w_P_x(integer x)
13   ...                        {
14 }                             fprintf(fwx,"%d",x);
15                               fprintf(fwx,"\n");
16                               fflush(fwx);
17                            }
```

6.1.2 Sequential code generation

In the context of this work, we shall consider the sequential, inlining code generation scheme for the step function that directly interprets the SIGNAL process obtained after the clock calculation, Boolean abstraction, and scheduling phases of the compiler front-end. We describe the code structure of the step function for a simple process represented in Listing 6.3. The step function obtained by compiling it is given in Listing 6.4. The C code introduces an explicit variable for each signal to represent the clock. Variable C_N is the clock of N and C_FB1, C_FB2 are the clocks of $FB1$ and $FB2$, respectively. As soon as the clock is evaluated and is `true`, the signal is read if it is an input signal or updated, otherwise. The precedence of the statements must be consistent with the graph of conditional dependencies, and one can observe the tree structure of conditional if-then-else statements which expresses directly the clock hierarchy.

The step function works as follows. It reads the clock values of $FB1$ and $FB2$. If C_FB2, the clock of $FB2$, has the value `true`, a new value for $FB2$ is read and used to compute the clock of N. In a similar way, if C_FB1 has the value `true`, a new value for $FB1$ is read. If C_N, the clock of N, has the value `true`, N gets the value $4 * FB1$. The updated value of N is also output.

A trace of this program is given below. At the initialization, the variables can have arbitrary values which are denoted by $*$.

```
1 FB1   *  1  2  2   3  5  4  6  9  ...
2 FB2   *  3  0  1   5  4  2  6  2  ...
3   N   *  4  4  4  12 20 20 24 24  ...
```

Taking into account that N in C code is the value of the corresponding signal in SIGNAL program, we have an observation that in a run of the step function, the value of N is unchanged when C_N, the clock of N, has the value `false`. Intuitively, based on this observation, we can say that if there is a variable in the generated C program whose value is never updated in a run of the step function. Then it will be assigned no value, denoted as $\perp$. In the next sections, we shall show how this assumption can be formalized to represent the computation of step function as a shared value-graph.

6.2 Illustrative example

We begin by showing how our verification process works for an illustrative example. Consider the synchronous program `WHENOP` written in SIGNAL language which is given in Listing 6.3.

Listing 6.3 Program WHENOP in SIGNAL

```
process WHENOP=
(? integer FB1; integer FB2;
 ! integer N)
(| N := 4*FB1 when (FB2>=3)
 |)
end;
```

`WHENOP_step` which is shown in Listing 6.4 is the *step function* of the generated C code. This function which is called repeatedly in an infinite loop, simulates one synchronous step of the SIGNAL program.

Listing 6.4 Synchronous Step of WHENOP

```
EXTERN logical WHENOP_step()
{
   if (!r_WHENOP_C_FB1(&C_FB1))
      return FALSE;
   if (!r_WHENOP_C_FB2(&C_FB2))
      return FALSE;
   if (C_FB2)
   {
      if (!r_WHENOP_FB2(&FB2))
         return FALSE;
   }
   C_CLK_36 = (C_FB2 ? (FB2 >= 3)
```

```
13                : FALSE);
14    C_N = C_FB1 && C_CLK_36;
15    if (C_FB1)
16    {
17       if (!r_WHENOP_FB1(&FB1))
18          return FALSE;
19    }
20    if (C_N)
21    {
22       N = 4 * FB1;
23       w_WHENOP_N(N);
24    }
25    WHENOP_step_finalize();
26    return TRUE;
27 }
```

In this example, we use the concept of *gated* ϕ-function such as $x = \phi(c, x_1, x_2)$ that is mentioned in more details in the next sections. It is used to represent a branching in a program, which means x takes the value of x_1 if the condition c is satisfied, and the value of x_2, otherwise. Since the generated C programs use *persistent* variables (i.e. variables that always have some values), while SIGNAL programs which use *volatile* variables. We shall assume that if a variable (including the input and output variables) in the generated C program is such that its value is never updated then it will be assigned the absent value, denoted as $\perp$. If a statement involves a variable x before its value update then the value of this variable is the previous value, denoted as $m.x$.

Considering the equation $N := 4 * FB1$ `when` $(FB2 >= 3)$, at a considered instant t, the signal N is present if the signals $FB1, FB2$ are present and $FB2$ is greater than or equal to 3. When N is present, its value is defined by the value of $FB1$ multiplied by 4. The value of N is $\perp$ when it is absent. The computation of this equation can be replaced by the following gated ϕ-function:

$$N = \phi(\widehat{N}, \tilde{N}, \perp)$$

where $\widehat{N} \Leftrightarrow (\widehat{FB1} \wedge \widehat{FB2} \wedge (\widetilde{FB2} >= 3))$, $\tilde{N} = 4 * \widetilde{FB1}$. Here, $\widehat{N}, \widehat{FB1}, \widehat{FB2}$ are Boolean variables that represent the states (`false`: absent, `true`: present) of signals $N, FB1$ and $FB2$ at instant t, respectively. And $\tilde{N}, \widetilde{FB1}, \widetilde{FB2}$ are values of signals $N, FB1$ and $FB2$ with the same types. Then, this gated ϕ-function indicates that at any instant t such that signals $FB1$ and $FB2$ are present and the value of $FB2$ is greater than or equal to 3, then the value of N is equal to the value of $FB1$ multiplied by 4, otherwise the value of N is $\perp$.

In the same way, we use a gated ϕ-function to represent the branching in C code. For instance, if C_N is `true` then the value of variable N is defined by the value of $FB1$ multiplied by 4. Otherwise, the value of N is never updated. This computation can be replaced by the following gated ϕ-function:

$$N = \phi(C\_N, 4 * FB1, \perp)$$

We replace the variables C_CLK_36 and C_N by their definition, and we obtain the synchronous data-flow value-graph for the output N that is presented in Figure 6.3. Notice that the compiler prefers to write $C\_CLK\_36 = (C\_FB2?(FB2 >= 3) : FALSE)$ instead of $C\_CLK\_36 = C\_FB2\&\&(FB2 >= 3)$, which can be represented by the following gated ϕ-function:

$$C\_CLK\_36 = \phi(C\_FB2, FB2 >= 3, \texttt{false})$$

The dashed arrows are not parts of the graph. They only mean that for each node, there is a set of labels that indicates which nodes of the graph correspond to which signals, clocks or variables in the programs. The SIGNAL program and its generated C program have been represented in the same graph. The nodes which represent the same structures (clocks, signals, variables and function symbols) have been reused. The unique occurrence of a reused node is said to be *shared*. For example, the nodes labeled $>=$ and $\perp$ are shared in the graph.

The values of input signals and their corresponding variables in the generated C code are represented by the same nodes in the shared graph. In general, it is safe to assume that the values of input signals and the corresponding variables in the generated C code are equal. Thus, in the shared graph, the input signal values $\widetilde{FB1}, \widetilde{FB2}$ and the variables $FB1, FB2$ in the C code are represented by the same nodes.

Suppose that we want to verify that the signal N in the `WHENOP` and `WHENOP_step` (denoted by N^c) will have the same value. This can be done by showing that they are represented by the same subgraph. That means that they are labels of the same node in the graph. In the following, we say that they point to the same node in the graph. In Figure 6.3, we cannot conclude that they are equivalent, however we can transform the value-graph by applying *normalization* rules. First, by exploiting the generated program, C_FB1, C_FB2 are clocks of $FB1$ and $FB2$, respectively (the values of inputs $FB1, FB2$ are updated only when C_FB1, C_FB2 are valid). The first rule we shall apply is that: *"If x is an input and the clock of x is read as input parameter (it is not defined in the program) then its clocks in* SIGNAL *program and C code are represented by the same node"*. Thus, $\widehat{FB1}$ and C_FB1

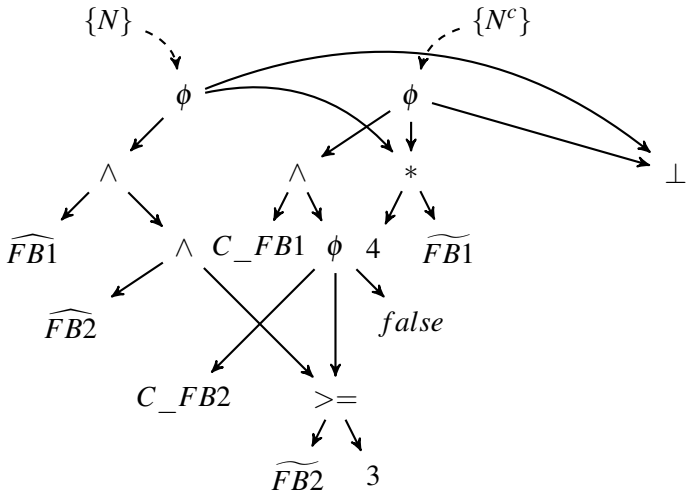

Fig. 6.3 The shared value-graph of WHENOP and WHENOP_step

are represented by the same node. With the same reason, $\widehat{FB2}$ and C_FB2 are represented by the same node as in Figure 6.4. Until now, the subgraphs that represent the variable N in the two programs are not rooted at the same node. We shall apply the following second rewrite rule to the resulting graph, and we shall replace $\phi(C\_FB2, \widetilde{FB2} >= 3, \texttt{false})$ with $C\_FB2 \wedge \widetilde{FB2}$

$$\phi(c, x, \texttt{false}) \textit{ is replaced by } (c \wedge x) \textit{ for any Boolean expression } x.$$

We shall go into details about the rules in the next section. After this replacement, and maximizing the variable sharing, the variable N in both programs points to the same node in the resulting graph (Figure 6.5). Therefore, we can conclude that the outputs are equivalent.

6.3 Synchronous data-flow value-graph

In this section, we describe the computation of a signal in a SIGNAL program and the corresponding variable in the generated C code in terms of SDVGs. Let us recall the computation of the output signal y in the equation $y := x$ `when` b in the previous section. At any instant, the signal y holds the value of x if the following conditions are satisfied:

- x and b are defined.
- b holds the value `true`.

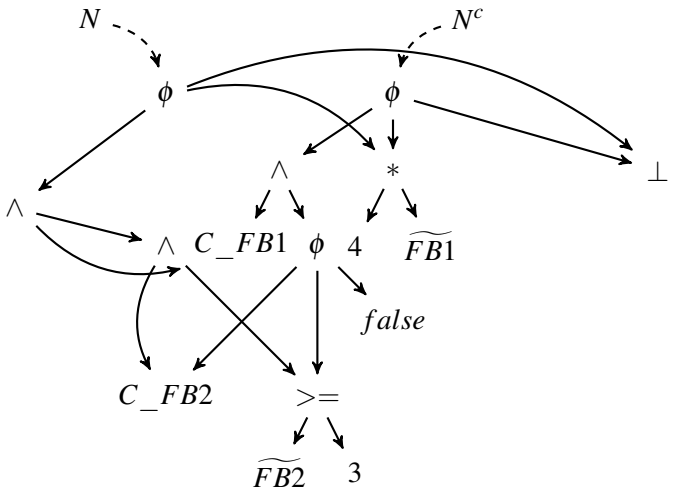

Fig. 6.4 The resulting transformed value-graph

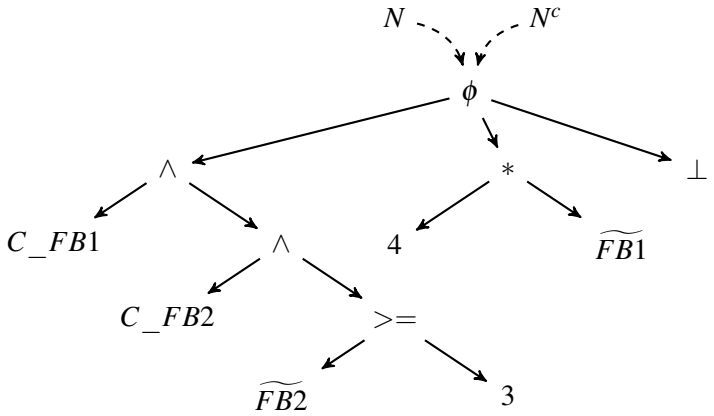

Fig. 6.5 The final value-graph

Otherwise, it holds no value. Thus, at a given instant, to represent the underlying control conditions and the computation of this equation, we can use the following gated ϕ-function:

$$y = \phi(\hat{x} \wedge \hat{b} \wedge \tilde{b}, \tilde{x}, \perp)$$

The condition $(\hat{x} \wedge \hat{b} \wedge \tilde{b})$ represents the state in which x holds a value ($\hat{x} = \texttt{true}$), and b holds the value `true` ($\hat{b} = \texttt{true} \wedge \tilde{b} = \texttt{true}$). This section explores a method to construct that shared value-graph for both SIGNAL and generated *C* code programs, which is the computational model of our translation validation approach.

6.3.1 Definition of SDVG

Graphs can be used to describe many structures in computer science: program control flows, communication processes, computer networks, pointer structure on the heap and many others. In fact, for most activities in software development, many types of visual notations have been introduced, including UML, state diagrams, control flows graphs, block diagrams. These notations construct models that can be seen as graphs. This section intends to focus on graphs which represent expressions for computing the variable values in programs. We present basic definitions, including the notion of gated ϕ-function, and introduce a linear syntax presentation for terms represented as graphs. Finally, we provide the definition of our considered type of graph, synchronous data-flow value-graph. The interested readers can refer to [53] for more detailed discussion on term graphs and linear syntax presentation for graphs.

6.3.1.1 Gated ϕ-function

In Static Single Assignment (SSA), a ϕ-node is placed at the confluence of a program control flow to represent the different choices of a variable. However, it does not contain the condition to determine which incoming branch reaching a confluence node is chosen. By contrast, *gating functions* are defined with some extra parameters to represent the conditions for choosing. To construct a SDVG, we shall employ the notation of gating function to capture the branching statements in computation of signals in synchronous data-flow programs and variables in the generated C code.

Gating functions were first introduced by Ballance et al. in [11] to represent the conditions that guard the paths to a ϕ-node. There are several types of gating functions as follows:

- The gated ϕ-function, which is an *if - then - else* representation. It captures the condition for choosing a branch of the confluence node. For instance, $x_3 = \phi(c, x_1, x_2)$ returns the value x_1 or x_2 depending on the value of c. If c is `true`, $x_3 = x_1$ and $x_3 = x_2$ if c is `false`.

- The μ function is used to capture the initial and loop-carried values at the header of a loop. For instance, $x_2 = \mu(i = 1, n, x_0, x_1)$ represents that x_2's initial value is x_0 when i is the first iteration and its subsequent value is x_1.

- The η function is placed at the loop exit. It selects the last value at the end of the loop. For instance, $x_2 = \eta(i > n, x_1)$ means that x_2 takes the last value computed by

the loop, x_1.

6.3.1.2 Terms as trees and graphs

Let X and F be an infinite set of *variables* and a (finite or infinite) set of *function symbols* such that $X \cap F = \emptyset$. Each $f \in F$, denoted by $f(x_1, x_2, ..., x_n)$, where n is the arity of f, has a number of arguments (or *arity*) greater or equal to 0. Function symbols of arity 0 are called *constants*. Then the set of *terms* T is inductively defined by the following rules:

- Any variable is a term.
- Any expression $f(t_1, t_2, ..., t_n)$, where $f \in F$ and $t_1, ..., t_n \in T$, is a term.

Given a term t, the *substerms* of t are t and if $t = f(t_1, ..., t_n)$, all subterms of $t_1, ..., t_n$. For example, let $X = \{x, y\}$ and $F = \{f, g\}$, then $T = \{x, y, f(x), g(y), f(x, y)...\}$. Note that we do not assume that function symbols have fixed arities.

Definition 29 *A directed term graph over X and F is a pair $\langle N, succ \rangle$ involving a (finite or infinite) set N of nodes which can be labeled by an element in $X \cup F$ and a function $succ : N \longrightarrow N^*$. The set of nodes $n_1, ..., n_k = succ(n)$ is the set of successors of n.*

We write $succ(n)_i$ to denote the i^{th} element of $succ(n)$. The pair of nodes $e = (n, succ(n)_i)$ is called an edge, the set of all edges is denoted by E. When we draw the visual representation of graphs, a directed edge e will go from n to $succ(n)_i$, the ordering of the edges from n to $succ(n)$ is left-to-right corresponding to the ordering of the elements of $succ(n)$.

For example, let $X = \{a, b, c\}, F = \{+, *\}$ be the set of variables and the set of function symbols, respectively. The set of nodes and the function *succ* is defined as follows:

$$N = \{+, *, a, b, c\}$$
$$succ(+) = (a, *), succ(*) = (b, c), succ(a) = (), succ(b) = (), succ(c) = ()$$

This defines a directed graph which is depicted on the left of Figure 6.6. Other directed graphs over F and X are depicted on the right side of the figure. Note that in the third graph, the nodes with repeated label b are represented by a same node, and the label is shared in the graph.

Given a directed graph $G = \langle N, succ \rangle$, a path in G is a list of nodes $(n_0, n_1, ..., n_k)$ where $k \geq 0$ and n_{i+1} is a successor of n_i. This path is said to be from n_0 to n_k and k is the length of the path. A path of length greater than 0 from a node to itself is called a *cycle*, and the repeated node is called *cyclic* node. A graph which contains a cycle is a *cyclic* graph, otherwise it is *acyclic*.

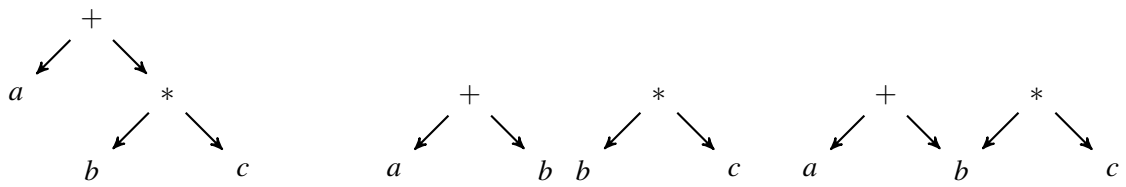

Fig. 6.6 The directed graphs of $a+b*c$ and $a+b, b*c$

Definition 30 *A tuple $\langle N, succ, r\rangle$ is a term graph at the node $r \in N$ where $\langle N, succ\rangle$ is a directed graph, if every node of the term graph is reachable by a path from r. The node r is called the root of the graph.*

In a term graph, a path from the root is said to be *rooted*. The term graph is *root-cyclic* if there is a cycle containing the root. Given a directed graph $G = \langle N, succ\rangle$, let n be a node in G. The subgraph of G rooted at n is the term graph $\langle N', succ', n\rangle$ where $N' = \{n' \in N |$ there is a path from n to $n'\}$ and $succ'$ is the restriction of $succ$ to N'. We write $G\,|_n = \langle N_{G|_n}, succ_{G|_n}\rangle$ to denote the fact that $G\,|_n$ is the subgraph rooted at n of G.

Consider, for example, the following directed graph which is depicted on the left of Figure 6.7. The subgraph rooted at the node labeled $+$ is depicted as the second graph in the figure. The third graph is a root-cyclic graph.

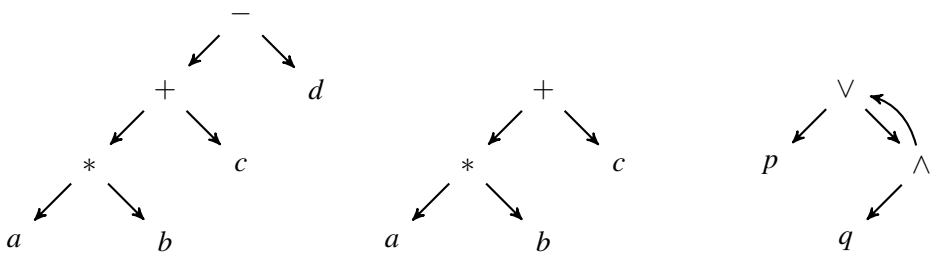

Fig. 6.7 The subgraph rooted at node labeled $+$ and a root-cyclic graph

The linear notation for graphs is defined by the following context-free grammar:

$$\begin{aligned} graph &:= node \mid node + graph \\ node &:= x \mid f(node,...,node) \mid nid \mid nid : x \mid nid : f(node,...,node) \end{aligned}$$

f and x are respectively a symbol function in F and a variable in X; nid ranges over a set, disjoint from X and F, of *node identifiers*. Any node identifier nid in a graph which identifies a node must occur exactly once in the context $nid : x$ or $nid : f(node,...,node)$. The constants are simply represented by symbol functions with arities equal to 0. This

syntax is similar to the syntax for graphs in [12]. For instance, the three graphs of the examples in Figure 6.6 can be expressed in this syntax as follows:

$$+(a,*(b,c)),+(a,b)+*(b,c) \text{ and } +(a,nid_1:b)+*(nid_1,c)$$

Note that multiple uses of the same node identifier mean that there are multiple references to the same node.

Definition 31 *A tree is a term graph at the node r such that there is exactly one path from r to each node in the graph.*

Based on the above definition, in Figure 6.7 the first and second graphs are trees and the third one is not. Thus it is obvious that a tree is always acyclic.

6.3.1.3 Homomorphisms of graphs and trees

Let $G_1 = \langle N_1, succ_1, r_1 \rangle$ and $G_2 = \langle N_2, succ_2, r_2 \rangle$ be two graphs, the definition of *homomorphism* is given as follows:

Definition 32 *An homomorphism from G_1 to G_2 is a map $f : N_1 \longrightarrow N_2$ such that for all $n \in N_1$,*

- *$f(n)$ and n have the same label*
- *$succ_2(f(n)) = f(succ_1(n))$*

where $f(n_1,...,n_k) = (f(n_1),...,f(n_k))$. This definition states that homomorphisms preserve node labels, successors and their order. We write $G_1 \longrightarrow G_2$ to denote the fact that there is an homomorphism from G_1 to G_2. Figure 6.8 shows an example of homomorphism.

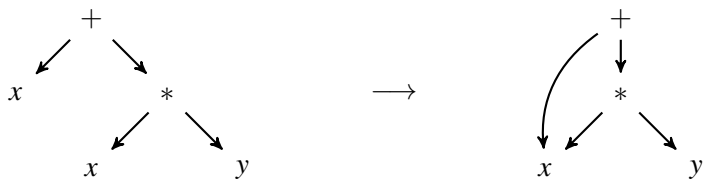

Fig. 6.8 An example of homomorphism

Definition 33 *Let $G_1 = \langle N_1, succ_1, r_1 \rangle$ and $G_2 = \langle N_2, succ_2, r_2 \rangle$ be two graphs.*

- *An homomorphism f from G_1 to G_2 is rooted if $f(r_1)$ and r_2 have the same label.*

- *An isomorphism is an homomorphism which is inverse. We denote an isomorphism from G_1 to G_2 by $G_1 \sim G_2$.*

- *When there is a rooted isomorphism from G_1 to G_2, we say that they are equivalent, denoted by $G_1 \approx G_2$.*

Proposition 6 *For all graphs G_1 and G_2, we have $G_1 \approx G_2$ implies $G_1 \sim G_2$. Every rooted homomorphism from one tree to another is an isomorphism.*

6.3.1.4 Synchronous data-flow value-graph

Let X be the set of all variables which are used to denote the signals, clocks and variables in a SIGNAL program and its generated C program. In our consideration, the functions which apply on signal values in the primitive *stepwise functions* are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`) and numerical operators (`+`, `-`, `*`, `/`). A constant is defined as a function symbol of arity 0. Thus in this chapter, we consider the set of function symbols which consists of the above functions and the gated ϕ-function, denoted by F.

As it is illustrated in Section 6.2, the computation of signals in a SIGNAL program and variables in the corresponding generated C code can be represented as a directed graph, in which a node can have multiple parents, and identical subgraphs are reused. That makes the maximal sharing among graph nodes. We shall consider the definition and examine some basic properties of SDVG. Formally, a SDVG is defined as follows:

Definition 34 *A* SDVG *associated with a* SIGNAL *program and its generated C code is a directed graph $G = \langle N, E, I, O, l_N, m_N \rangle$, where:*

- *N is a finite set of nodes.*

- *$E \subseteq N \times N$ is the set of edges. It describes the computation relation between the nodes.*

- *$I \subseteq N$ is the set of input nodes. They are the input signals and their corresponding variables in the generated C code.*

- *$O \subseteq N$ is the set of output nodes. They are the output signals and their corresponding variables in the generated C code.*

- *$l_N : N \longrightarrow X \cup F$ is a mapping labeling each node with an element in $X \cup F$. A node represents a clock, a signal, a variable, an operator or a gated ϕ-node function. And the subgraph rooted at a node is used to describe the computation of the corresponding element labeled at the node.*

- *$m_N : N \longrightarrow \mathscr{P}(V)$ is a mapping labeling each node with a finite set of clocks, signals, and variables. It defines the set of clocks, signals or variables in the* SIGNAL *program and the generated C code such that they are equivalent to the label of the node.*

In the rest of this chapter, we denote the fact that there exists an edge between two nodes x and y by $x \longrightarrow y$. A *path* from x to y is any set of nodes $s = \{x_0, x_1, ..., x_k\}$ such that $\forall i = 0, .., k-1, x_i \longrightarrow x_{i+1}$. An edge is a special case when $k = 1$. In a shared value-graph, the set of clocks, signals or variables $\{x_0, ..., x_n\}$ that they are equivalent to the node labeled by y is written as a node with label $\{x_0, ..., x_n\}\ y$.

6.3.2 SDVG of SIGNAL programs

Let P be a SIGNAL program, we write $X = \{x_1, ..., x_n\}$ to denote the set of all signals in program P which consists of input, output, state (corresponding to delay operator) and local signals, denoted by I, O, S and L, respectively. For each $x_i \in X$, we use $\mathbb{D}_{x_i}$ to denote its domain of values, and $\mathbb{D}_{x_i}^{\perp} = \mathbb{D}_{x_i} \cup \{\perp\}$ to denote its domain of values with the absent value, where $\perp \notin \mathbb{D}_{x_i}$. Then, the domain of values of X with absent value is defined as follows:

$$\mathbb{D}_X^{\perp} = \bigcup_{i=1}^{n} \mathbb{D}_{x_i} \cup \{\perp\}$$

With each signal x_i (Boolean or non-Boolean type), we attach a Boolean variable $\widehat{x_i}$ to encode its clock at a given instant (`true`: x_i is present, `false`: x_i is absent), and $\widetilde{x_i}$ with the same type as x_i to encode its value. Formally, the abstract values to represent the abstract clock and value of a signal can be represented by a gated ϕ-function, $x_i = \phi(\widehat{x_i}, \widetilde{x_i}, \perp)$, where $\widehat{x_i}$ and $\widetilde{x_i}$ are computed using the following functions:

$\hat{}: X \longrightarrow \mathbb{B}$ associates a signal with a Boolean value,
$\tilde{}: X \longrightarrow \mathbb{D}_X$ associates a signal with a value of same type as the signal.

Assume that the computation of signals in processes P_1, P_2 is represented as shared value-graphs G_1 and G_2, respectively. Then the value-graph G of the synchronous combination process $P_1|P_2$ can be defined as $G = \langle V, E, I, O, m_N \rangle$ in which for any node x, we replace it

by the subgraph that is pointed by x in G_1, G_2. And in G_1 and G_2, every identical subgraph is reused, in other word, we maximize sharing among graph nodes in G_1 and G_2. Thus, the shared value-graph of a SIGNAL program P can be constructed as a combination of the sub-value-graphs of its equations as above.

To demonstrate the above combination rule, we consider a simple SIGNAL program P as follows:

Listing 6.5 Simple Program in SIGNAL

```
process P=
(? integer x;
 ! integer y)
(| y := x * x1
 | x1 := x + 1
|)
where integer x1
end;
```

Suppose that we have the subgraphs that represent the equations $y := x * x1$ and $x1 := x + 1$ as depicted in Figure 6.9 (here, we omit to represent the abstract clocks of x and $x1$ at the node $\hat{y}$, $\widetilde{x1}$ at the node $+$). We replace the node $x1$ by the subgraph which defines it while reusing the identical node x. As a result, the shared synchronous data-flow value-graph of P is depicted in Figure 6.10.

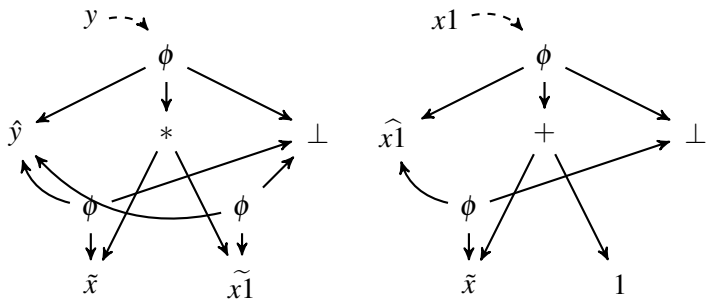

Fig. 6.9 The subgraphs of $y := x * x1$ and $x1 := x + 1$

A SIGNAL program is built through the set of primitive operators. Therefore, it is obvious that to construct SDVGs of SIGNAL programs, we shall construct a subgraph for each primitive operator. In the following, we present the value-graph corresponding to each SIGNAL primitive operator.

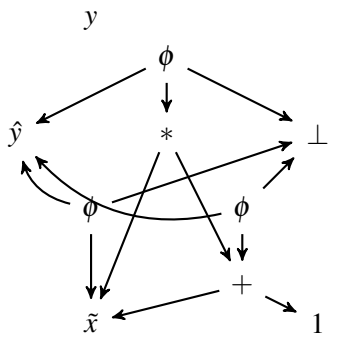

Fig. 6.10 The SDVG graph of P

6.3.2.1 Stepwise functions

Consider the operator $y := f(x_1,...,x_n)$, it indicates that if all signals from x_1 to x_n are defined, then the output signal y is defined by the result of the function f on the values of $x_1,...,x_n$. Otherwise, it is assigned no value. Thus, the computation of y can be represented by the following gated ϕ-function:

$$y = \phi(\hat{y}, f(\widetilde{x_1},\widetilde{x_2},...,\widetilde{x_n}), \perp)$$

where

$$\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2} \Leftrightarrow ... \Leftrightarrow \widehat{x_n}$$

The synchronous data-flow value-graph of the *stepwise functions* is depicted in Figure 6.11. Note that in the graph, $\{\widehat{x_1},...,\widehat{x_n}\},\hat{y}$ means that the subgraph representing the computation of $\hat{y}$ is also the computation of $\widehat{x_1},...,$ and $\widehat{x_n}$. In other words, $\widehat{x_1},...,\widehat{x_n}$ and $\hat{y}$ are equivalent, meaning that they label the same node in the graph.

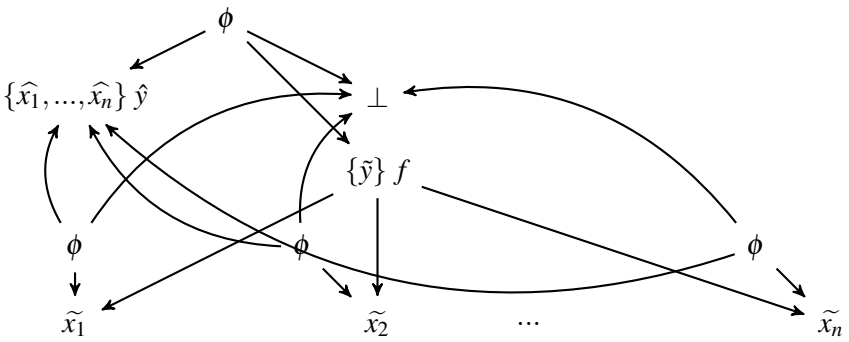

Fig. 6.11 The graph of $y := f(x_1,...,x_n)$

For instance, consider the following SIGNAL equation:

$$y := (x >= 1) \text{ and } c$$

It can be represented by the gated ϕ-function, $y = \phi(\hat{y}, \widetilde{x_1} \wedge \tilde{c}, \perp)$, where we replace $(x >= 1)$ by the fresh signal x_1. Thus the graphic representation is depicted in Figure 6.12.

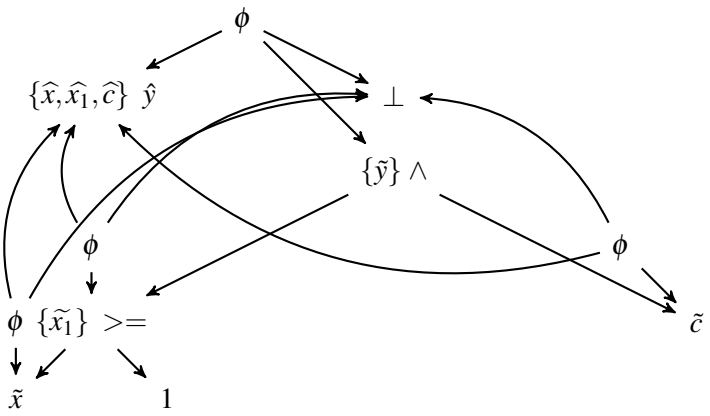

Fig. 6.12 The graph of $y := (x >= 1)$ and c

6.3.2.2 Delay

Consider the basic process which corresponds to the *delay* operator $y := x\$1$ `init` a. The output signal y is defined by the last value of the signal x when the signal x is present. Otherwise, it is assigned no value. Thus, the computation of y can be represented by the following nodes:

$$y = \phi(\hat{y}, \widetilde{m.x}, \perp) \text{ and } \widetilde{m.x_0} = a$$

where

$$\hat{y} \Leftrightarrow \hat{x}$$

$\widetilde{m.x}$ and $\widetilde{m.x_0}$ are the last value of x and the initialized value of y. The synchronous data-flow value-graph of the *delay* operator is depicted in Figure 6.13. Note that in the graph, $\{\hat{x}\}$ $\hat{y}$ means that the clocks $\hat{x}$ and $\hat{y}$ are equivalent, meaning that they point to the same node in the graph.

For instance, consider the following SIGNAL equation:

$$y := (x\$1 \texttt{ init } 1) + z$$

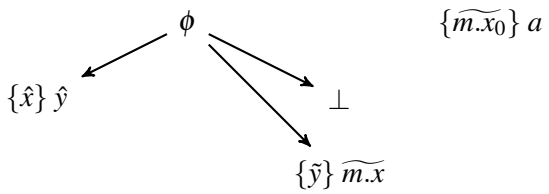

Fig. 6.13 The graph of $y := x\$1$ init a

It can be represented by the gated ϕ-function, $y = \phi(\hat{y}, \widetilde{m.x} + \tilde{z}, \perp)$ and the node $\widetilde{m.x_0} = 1$. Thus the graphic representation is depicted in Figure 6.14.

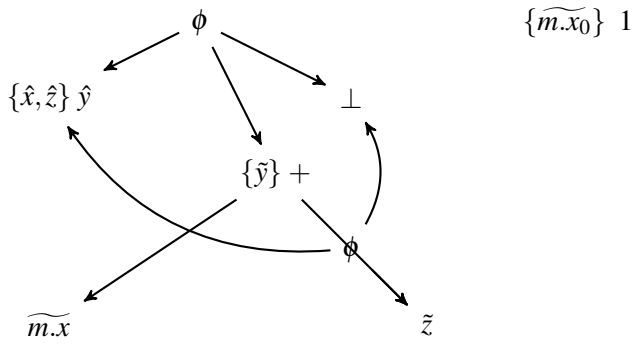

Fig. 6.14 The graph of $y := (x\$1 \text{ init } 1) + z$

6.3.2.3 Merge

Consider the basic process which corresponds to the *merge* operator $y := x$ `default` z. If the signal x is defined then the signal y is defined and holds the value of x. The signal y is assigned the value of z when the signal x is not defined and the signal z is defined. When both x and z are not defined, y holds no value. The computation of y can be represented by the following node:

$$y = \phi(\hat{y}, \phi(\hat{x}, \tilde{x}, \tilde{z}), \perp)$$

where

$$\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$$

The representation uses a nested ϕ-function which indicates that when $\hat{y}$ is `true`, y is defined by the gated ϕ-function $\phi(\hat{x}, \tilde{x}, \tilde{z})$. Otherwise, it holds no value. The synchronous data-flow value-graph of the *sampling* operator is depicted in Figure 6.15. Note that in the graph, the clock $\hat{y}$ is represented by the subgraph of $\hat{x} \vee \hat{z}$.

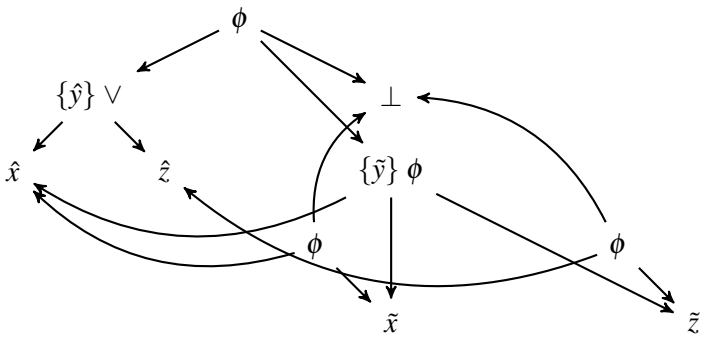

Fig. 6.15 The graph of $y := x$ default z

For instance, consider the following SIGNAL equation:

$$y := x \texttt{ default } (z+1)$$

It can be represented by the nested ϕ-function, $y = \phi(\hat{y}, \phi(\hat{x}, \tilde{x}, \widetilde{z_1}), \perp)$, where we replace $(z+1)$ by the fresh signal z_1. Thus the graphic representation is depicted in Figure 6.16.

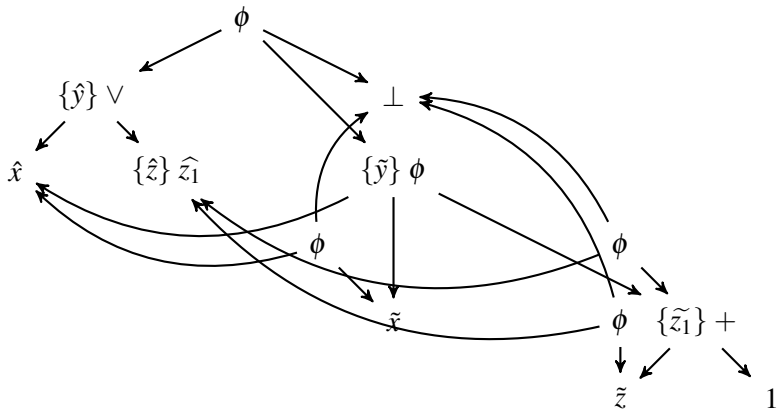

Fig. 6.16 The graph of $y := x$ default $(z+1)$

6.3.2.4 Sampling

Consider the basic process which corresponds to the *sampling* operator $y := x$ `when` b. If the signal x, b are defined and b holds the value `true`, then the signal y is defined and holds the value of x. Otherwise, y holds no value. The computation of y can be represented by the following node:

$$y = \phi(\hat{y}, \tilde{x}, \perp)$$

where

$$\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \tilde{b})$$

The synchronous data-flow value-graph of the *sampling* operator is depicted in Figure 6.17. Note that in the graph, the clock $\hat{y}$ points to the root of the subgraph of $(\hat{x} \wedge \hat{b} \wedge \tilde{b})$.

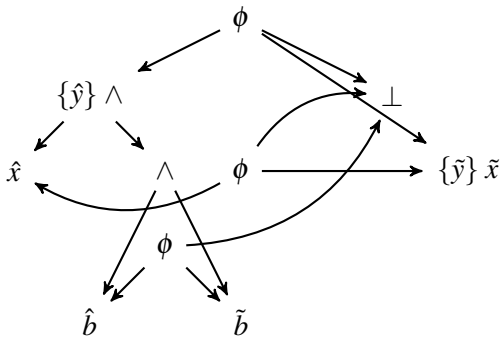

Fig. 6.17 The graph of $y := x$ when b

For instance, consider the following SIGNAL equation:

$$y := x \texttt{ when } (z >= 1)$$

It can be represented by the gated ϕ-function, $y = \phi(\hat{y}, \tilde{x}, \perp)$, where $\hat{y} \Leftrightarrow (\hat{x} \wedge \widehat{z_1} \wedge \widetilde{z_1})$ and we replace $(z >= 1)$ by the fresh signal z_1. Thus the graphic representation is depicted in Figure 6.18.

6.3.2.5 Restriction

The shared synchronous data-flow value-graph of *restriction* process $P_1 \backslash x$ is the same as the graph of P_1.

6.3.2.6 Clock relations

Given the above graph representations of the primitive operators, we can obtain the shared value-graphs for derived operators on clocks as depicted in Figure 6.19 and Figure 6.20. Here, z is a signal of type *event*. Its computation can be represented by the following gated ϕ-function:

$$z = \phi(\hat{z}, \texttt{true}, \perp)$$

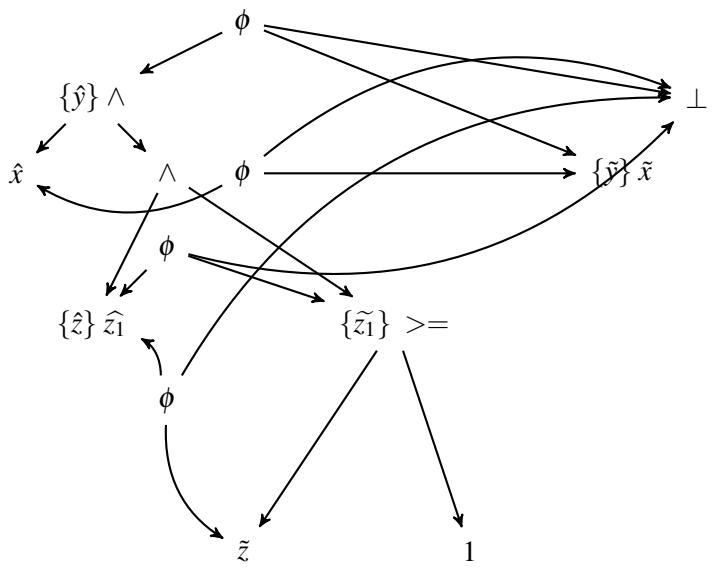

Fig. 6.18 The graph of $y := x$ when $(z >= 1)$

The clock relations between signals are given as follows:

$$
\begin{aligned}
z := \hat{x}: &\quad \hat{z} \Leftrightarrow \hat{x} \\
x \hat{}= y: &\quad \hat{x} \Leftrightarrow \hat{y} \\
z := x \hat{}+ y: &\quad \hat{z} \Leftrightarrow (\hat{x} \vee \hat{y}) \\
z := x \hat{}* y: &\quad \hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y}) \\
z := x \hat{}- y: &\quad \hat{z} \Leftrightarrow (\hat{x} \wedge \neg \hat{y}) \\
z := \texttt{when}\ b: &\quad \hat{z} \Leftrightarrow (\hat{b} \wedge \tilde{b})
\end{aligned}
$$

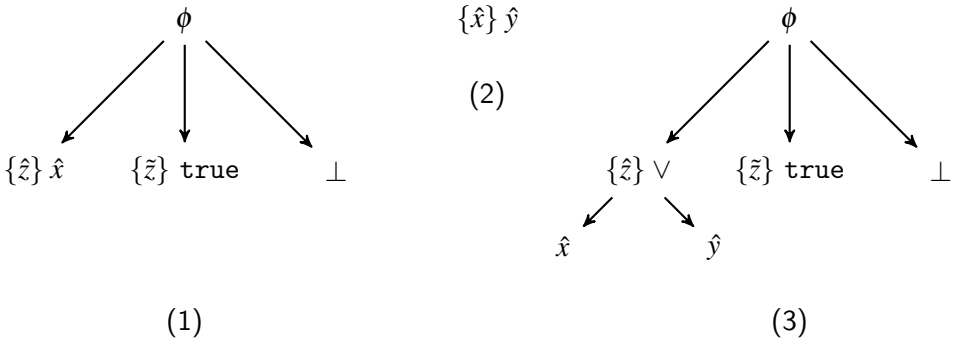

Fig. 6.19 The graphs of (1) $z := \hat{x}$, (2) $x \hat{}= y$ and (3) $z := x \hat{}+ y$

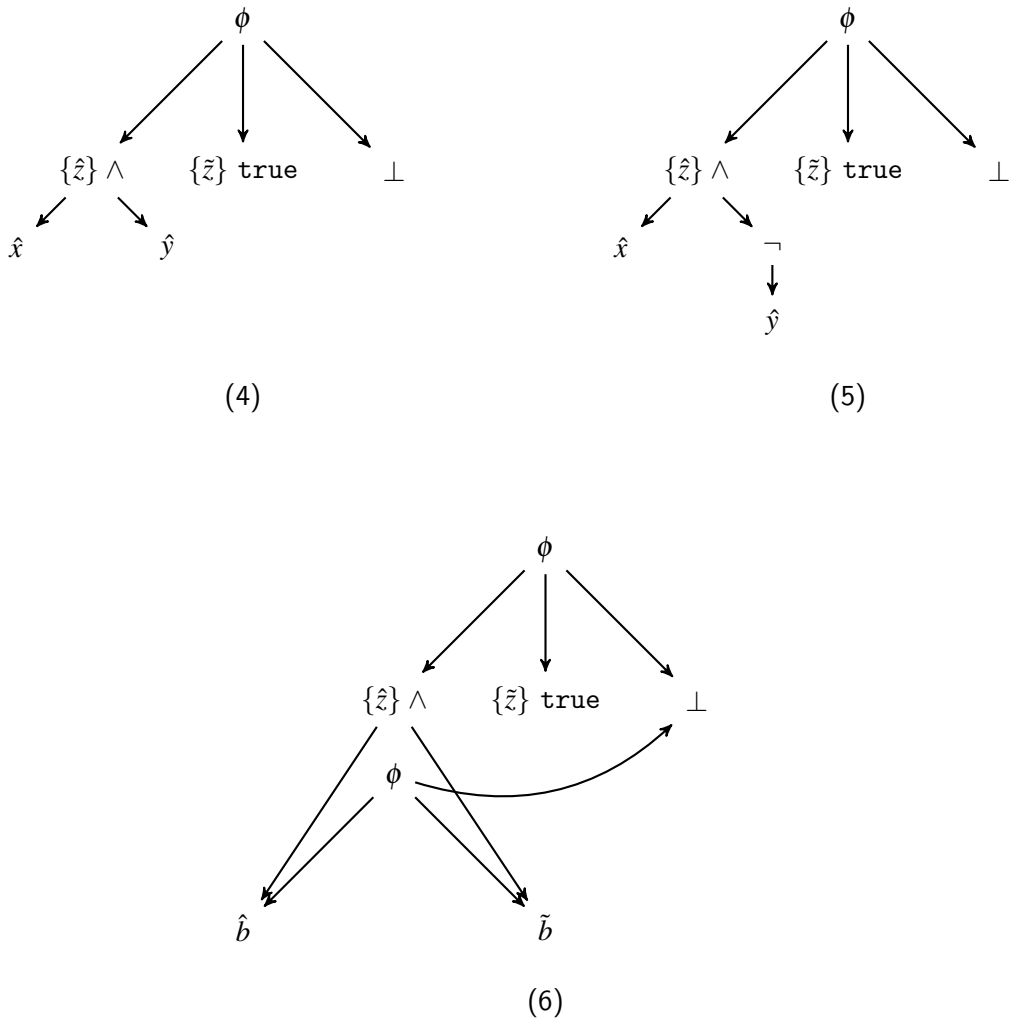

Fig. 6.20 The graphs of (4) $z := x \hat{*} y$, (5) $z := x \hat{-} y$ and (6) $z :=$ when b

6.3.3 SDVG of generated C code

For constructing the shared value-graph, the generated C program is translated into a sub value-graph along with the sub value-graph of the SIGNAL program. Let A be a SIGNAL program and C its generated C code, we write $X_A = \{x_1, ..., x_n\}$ to denote the set of all signals in A, and $X_C = \{x_1^c, ..., x_m^c\}$ to denote the set of all variables in C. We added "c" as superscript for the variables, to distinguish them from the signals in the SIGNAL program.

As described in the principles of code generation in the SIGNAL compiler, the generated C program of the SIGNAL program A consists of the following files:

- *A_main.c* is the implementation of the *main function*. This function opens the IO communication channels, and calls the *initialization function*. Then it calls the *step*

function repeatedly in an infinite loop to interact with the environment.

- *A_body.c* is the implementation of the *initialization function* and the *step function*. The initialization function is called once to provide initial values to the program variables. The step function, which contains also the step initialization and finalization functions, is responsible for the calculation of the outputs to interact with the environment. This function, which is called repeatedly in an infinite loop, is the essential part of the concrete code.

- *A_io.c* is the implementation of the *IO communication functions*. The IO functions are called to setup communication channels with the environment.

The scheduling and the computations are done inside the step function of the generated C program. Therefore, it is natural to construct a sub value-graph of this function in order to prove that its variables and the corresponding signals have the same values. To construct the value-graph of the *step function*, the following considerations need to be studied.

An original signal named x has a corresponding Boolean variable named *C_x* in the *step function*. Then the computation of x is implemented by conditional *if - then -else* statements as follows:

```
1 if (C_x)
2 {
3    computation(x);
4 }
```

If x is an input signal then its computation is the reading operation which gets the value of x from the environment. In case x is an output signal, after computing its value, it will be written to the IO communication channel with the environment. Note that the C programs use persistent variables (e.g. variables which always have some value) to implement the input SIGNAL programs which use volatile variables. As a result, there is a difference in the types of a signal in a SIGNAL program and of the corresponding variable in the C program. When a signal has the absent value, $\perp$, at a given instant, the corresponding C variable always has a value. For instance, in the *step function* `WHENOP_step()`, we observe that the value of variable *N* is maintained when the Boolean variable *C_N* has the value `false` as in the following code segment:

```
1 if (C_N)
2 {
3    N = 4 * FB1;
4    w_WHENOP_N(N);
5 }
```

This consideration implies that we have to detect that a variable in the C program whose value is never updated. In this case, it will be assigned the absent value, $\perp$. Thus, the computation of such variables, called x^c, can fully be represented by a gated ϕ-function as follows:

$$x^c = \phi(C\_x^c, \widetilde{x^c}, \perp)$$

where $\widetilde{x^c}$ denotes the newly updated value of the variable. For example, we consider the generated code of basic process corresponding to the primitive operator *merge* in Listing 6.6.

Listing 6.6 SDVGMerge in SIGNAL

```
process SDVGMerge=
(? integer x, y; boolean cx, cy;
 ! integer z;
)
(| z := x default y
 | x ^= when cx
 | y ^= when cy
 |)
;
```

The generated C code of the *step function* is given in Listing 6.7. The computation of variable z in this function can be represented by the following gated ϕ-function:

$$z^c = \phi(C\_z^c, \phi(cx\_50^c, \widetilde{x^c}, \widetilde{y^c}), \perp)$$

The shared value-graph of `SDVGMerge_step()` is depicted in Figure 6.21. In this graph, we replace the node $\phi(C\_cx^c, \phi(\widetilde{cx^c}, \widetilde{x^c}, \perp), \perp)$ by $\phi(C\_cx^c \wedge \widetilde{cx^c}, \widetilde{x^c}, \perp)$. We do the same for the node $\phi(C\_cy^c, \phi(\widetilde{cy^c}, \widetilde{y^c}, \perp), \perp)$.

Listing 6.7 Generated C code of SDVGMerge

```
EXTERN logical SDVGMerge_step()
{
   if (!r_SDVGMerge_C_cx(&C_cx)) return FALSE;
   if (!r_SDVGMerge_C_cy(&C_cy)) return FALSE;
   if (C_cx)
   {
      if (!r_SDVGMerge_cx(&cx)) return FALSE;
      if (cx)
      {
```

```
10          if (!r_SDVGMerge_x(&x)) return FALSE;
11       }
12    }
13    cx_50 = (C_cx ? cx : FALSE);
14    if (C_cy)
15    {
16       if (!r_SDVGMerge_cy(&cy)) return FALSE;
17       if (cy)
18       {
19          if (!r_SDVGMerge_y(&y)) return FALSE;
20       }
21    }
22    cy_56 = (C_cy ? cy : FALSE);
23    C_z = cx_50 || cy_56;
24    if (C_z)
25    {
26       if (cx_50) z = x; else z = y;
27       w_SDVGMerge_z(z);
28    }
29    SDVGMerge_step_finalize();
30    return TRUE;
31 }
```

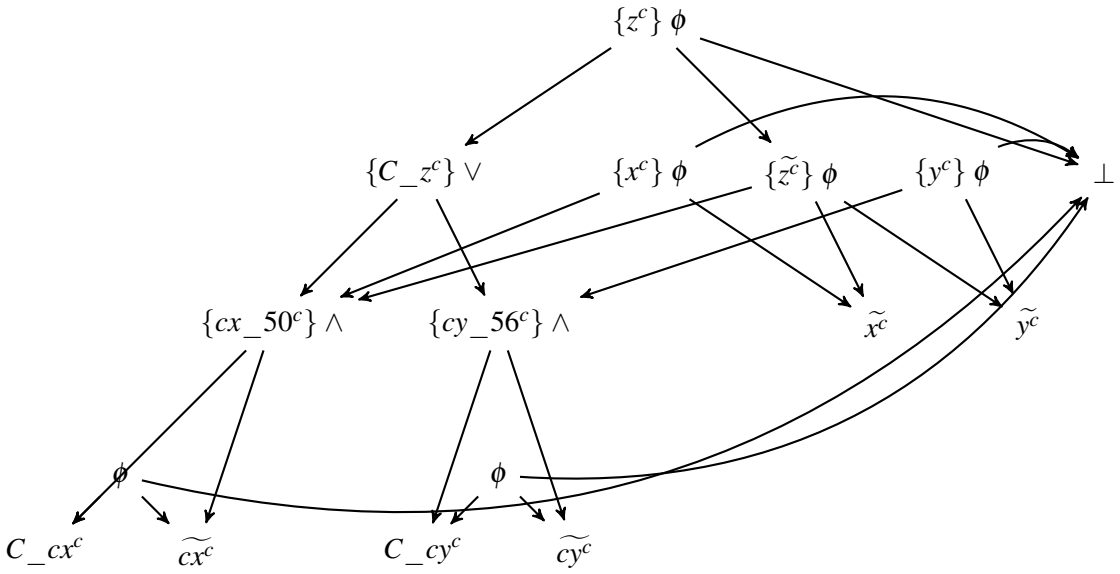

Fig. 6.21 The graph of SDVGMerge_step

In the generated C program, the computation of the variable whose clock is the master clock, which ticks every time the *step function* is called, is implemented without the condi-

tional *if - then -else* statement and it is always updated when the *step function* is invoked. The computation of such variables can be represented by a node in the shared value-graph as follows:

$$\{\widetilde{x^c}\}\ x^c$$

That means that the the variable x^c is always updated and holds the value $\widetilde{x^c}$. For instance, we consider the following C code segment, it is generated from the program DEC in Listing 3.3 which computes the values of the output signal N.

```
1 if (C_FB) N = FB;
2 else N = N - 1;
3 w_DEC_N(N);
```

The computation of N can be represented by the sub value-graphs in Figure 6.22, where $m.N^c$ denotes the previous value of the variable N.

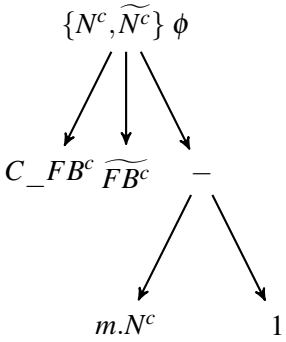

Fig. 6.22 The graph of N's computation

Considering the following code segment, the observation is that the variable x is involved in the computation of the variable y before the updating of x.

```
1 if (C_y)
2 {
3    y = x + 1;
4 }
5 ...
6 if (C_x)
7 {
8    x = ...
9 }
10 ...
```

In this situation, we refer to the value of x as the previous value, denoted by $m.x^c$. It can happen when the *delay* operator is applied on the signal x in the SIGNAL program. The

computation of y is represented by the following gated ϕ-function:

$$y^c = \phi(C\_y^c, m.x^c + 1, \perp)$$

6.4 SDVG translation validation

In this section, we introduce the set of rewrite rules to transform the shared value-graph resulting from the previous step. This procedure is called *normalizing*. At the end of the normalizing procedure, for any output signal x and its corresponding variable x^c in the generated C code, we check whether x and x^c point to the same node in the graph. That means they are represented by the same subgraph, meaning they have the same values. We also provide a method to implement the representation of synchronous data-flow value-graph and adapt the normalizing procedure with any future optimization of the compiler.

We introduce the graph rewriting techniques in Section 6.4.1. Section 6.4.2 provides the set of rewrite rules which is used to perform the normalizing procedure on the shared value-graph. In Section 6.4.3, we consider a method to implement the data structure of synchronous data-flow value-graph and the normalizing procedure.

6.4.1 An introduction to graph rewriting

Graphs provide a simple and powerful approach to a variety of problems of software engineering, and system modeling in particular. These graphs are mostly static descriptions of system states. Adding dynamics requires some ways to express state changes and thus graph transformations are involved, either explicitly or behind the scenes. In fact, the theory of graph transformation has found many applications in implementing functional languages based on term rewriting [136], in modeling of concurrent systems [54] and in other areas such as software engineering, hardware designs and visual languages.

First, we recall the basic concepts of term rewriting, and we explain informally how it works in a simple example. We then define our notion of graph rewriting. For a comprehensive introduction of term rewriting and graph rewriting, the interested reader may consult the textbook [53]. Consider, for example, the following rewrite rules for definition of natural number multiplication:

$$\begin{aligned} x*0 &\longrightarrow 0 \\ x*(y+1) &\longrightarrow (x*y)+x \end{aligned}$$

We shall apply the second rewrite rule to the expression $t*(u+1)$, where t and u are subex-

pressions, which is represented by the graph on the left of Figure 6.23. When applying the

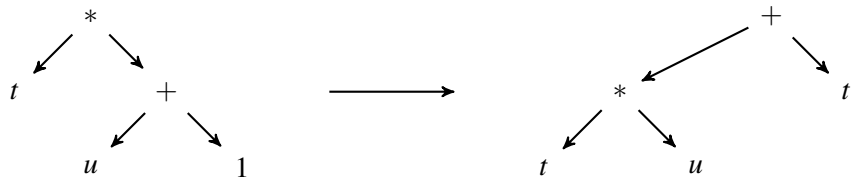

Fig. 6.23 The transformation of the graph of $t * (u + 1)$

above rewrite rule, the subexpression t is represented two times in the graph as depicted on the right of the figure. The subexpression t is evaluated two times if it has not been yet evaluated when we evaluate the expression. To solve this problem, an easy solution is, instead of copying the subexpression t, to create two references to the existing subexpression t, meaning that the repeated subterms are shared. Then the result of applying the above rewrite rule is given in Figure 6.24.

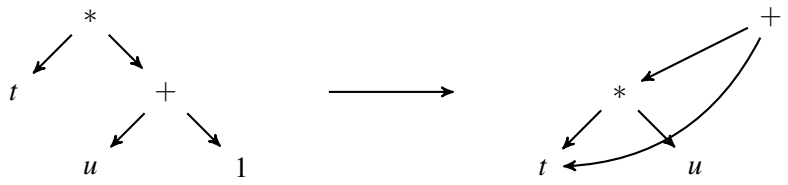

Fig. 6.24 The transformation of graph of $t * (u + 1)$ with sharing of repeated subterms

Let X and F be the set of variables and function symbols, such that $X \cap F = \emptyset$. We denote the set of all terms over X and F by $T_{X,F}$. A mapping $\sigma : T_{X,F} \longrightarrow T_{X,F}$ is called a *substitution* if $\sigma(c) = c$ and $\sigma(f(t_1,...,t_n) = f(\sigma(t_1),...,\sigma(t_n))$ for every constant c and term $f(t_1,...,t_n)$.

Definition 35 *A term rewrite rule over $T_{X,F}$ is a pair of terms (t_l, t_r), written as $t_l \rightarrow t_r$, such that:*

- *t_l is not a variable, and*
- $\forall x \in X.(x \in t_r \Rightarrow x \in t_l)$.

The terms t_l and t_r are, respectively, the left and right-hand sides of the rewrite rule. A rewrite rule is *left-linear* (resp. *right-linear*) if no variable occurs more than once in its left-hand side (resp. right-hand side).

A *term rewriting system* is a tuple $\langle F, R\rangle$, in which F is a set of function symbols and R is the set of term rewrite rules over $T_{X,F}$. A term rewriting system is said to be left-linear (resp. right-linear) if all its rules are.

Given two terms t_1 and t_2, we say that there exists a rewrite relation on $T_{X,F}$ for (t_1, t_2) induced by $\langle F, R\rangle$, denoted by $t_1 \rightsquigarrow t_2$, if it satisfies:

- There exists a rule $(l, r) \in R$ and a substitution σ such that $\sigma(l)$ is a subterm of t_1.
- The term t_2 is obtained from t_1 by replacing the occurrence of $\sigma(l)$ by $\sigma(r)$.

We now define the application of term rewriting rules to graph, in other words, we make the graph representation of term rewriting rules. Let $G = \langle N_G, succ_G, r_G\rangle$ be a graph and n and n' be nodes of G, a triple (G, n, n') is called a *graph rewrite rule* and n, n' are the *left root* and the *right root* of the rule.

Given a graph $G_0 = \langle N_0, succ_0, r_0\rangle$, a pair $\Delta = (r, f)$ is a *redex* in G_0 if r is a graph rewrite rule (G, n, n') and f is an homomorphism from $G\,|_n$ to G_0. The homomorphism is called an *occurrence* of the rule r. We first begin with some example to illustrate the definition of graph rewriting technique by showing how the translation of term rewrite rules to graph rewrite rules works.

Let $t_l \to t_r$ be a term rewrite rule, we shall construct the corresponding graph rewrite rule (G, n, n'). The construction works as follows. First we take the graphs representing both left and right hand sides of the term rewrite rule to form the union of these graphs, sharing those nodes which represent the same structures in t_l and t_r. This resulting shared graph is G. Then we take the roots of t_l and t_r to be n and n', respectively. For example, we illustrate the construction of the graph rewrite rule for the following term rewrite rule:

$$\phi(c, x, \texttt{false}) \to c \wedge x$$

We first make the graph representing of $\phi(c, x, \texttt{false})$ and $c \wedge x$ on the left of the Figure 6.25. Then the union graph sharing the same nodes (c and x) is depicted on the right. Finally, the left root n and right root n' are the nodes labeled ϕ and $\wedge$. Thus the graph rule is represented as follows:

$$(n : \phi(n_c : c, n_x : x, \texttt{false}) + n' : \wedge(n_c, n_x), n, n')$$

Let $((G, n, n'), f : G|_n \to G_0)$ be a redex in the graph G_0. We now present a formal definition of the general construction of the graph rewriting from a graph rewrite rule (G, n, n'). The construction consists of three phases: *build*, *redirection* and *garbage collection*, which are defined as follows.

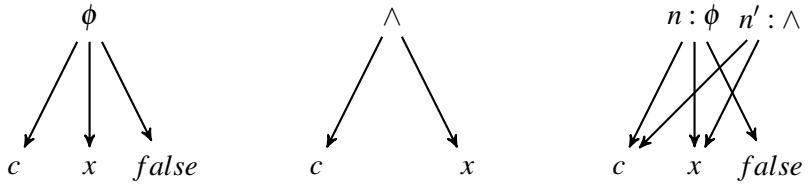

Fig. 6.25 The graph rule of the term rule $\phi(c,x,false) \rightarrow c \wedge x$

The build phase The resulting graph $G_1 = \langle N_1, succ_1, r_1 \rangle$ in this phase, denoted by $G_1 = G_0 +_f (G,n,n')$, is constructed as follows.

- $N_1 = N_0 \uplus (N_{G|_{n'}} \setminus N_{G|_n})$, where $\uplus$ denotes the disjoint union of two sets. For any node $m \in N_1$, it has the same label as in G_0 if $m \in N_0$. Otherwise, it has the same label as in G.
- $r_1 = r_0$.
- for every node $m_i = succ_1(m)_i$,

$$m_i = \begin{cases} succ_0(m)_i & \text{if } m \in N_0 \\ succ_G(m)_i & \text{if } m, succ_G(m)_i \in N_{G|_{n'}} \setminus N_{G|_n} \\ f(succ_G(m)_i) & \text{if } m \in N_{G|_{n'}} \setminus N_{G|_n}, succ_G(m)_i \in N_{G|_n} \end{cases}$$

The redirection phase In this step all references to the node $f(n)$ are replaced by the references to the node n', the resulting graph is denoted by $G_2 = \langle N_2, succ_2, r_2 \rangle = G_1[f(n) := n']$. This replacement is defined by a substitution of the node $f(n)$ for the node n as follows:

- for every node $c \in N_1$ and $c \in N_2$, they have the same label.
- for every node $c \in N_1$, $succ_2(c)_i = n'$ if $succ_1(c)_i = f(n)$, otherwise $succ_2(c)_i = succ_1(c)_i$.
- $r_2 = n'$ if $r_1 = f(n)$, otherwise $r_2 = r_1$.

The garbage collection phase In this last step, we define $G_3 = G_2|_{r_2}$ which is a part of the graph G_2 accessible from its root. We denote the graph G_3 by $GC(G_2)$.
Given a redex $\Delta = (r,f)$ and a graph G_0, the construction of the graph resulting from reducing the redex Δ in the graph G_0, denoted by $RED(\Delta, G_0)$, is defined as:

$$RED(((G,n,n')f), G_0) = GC((G_0 +_f (G,n,n'))[f(n) := n'])$$

To illustrate the above construction of graph rewriting, we consider a graph rewriting rule (G,n,n'), a graph G_0, and an homomorphism f from $G|_n$ to G_0 as depicted in Figure 6.26. The graph rewriting rule (G,n,n') and G_0 are given as follows:

$$(G,n,n') = (n : \phi(n_c : c, n_x : x, \texttt{false}) + n' : \wedge(n_c, n_x), n, n')$$
$$G_0 = \vee(\phi(c,x,\texttt{false}), y)$$

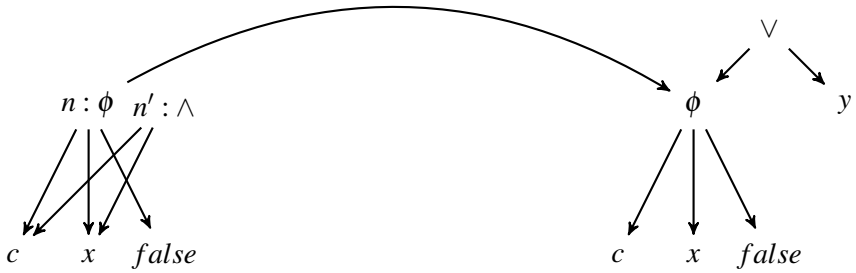

Fig. 6.26 An example of graph rewriting

We first copy the part of $G|'_n$ which is not contained in $G|_n$ to G_0, with node labels, successors, and root defined in the build phase. We obtain the resulting graph G_1. Then all edges of G_1 pointing to $f(n)$ are replaced by edges pointing to the copy of n', giving the graph G_2 as depicted in Figure 6.27. The root of G_2 is the root of G_1 if that node is not equal to $f(n)$. Otherwise, the root of G_2 is the copy of n' as described in the redirection phase. From the graph G_2, we remove all nodes which are not accessible from the root, giving the

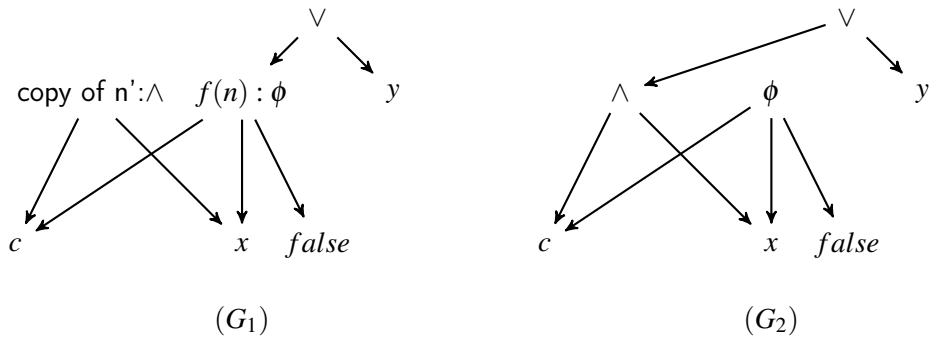

Fig. 6.27 Graph rewriting: Build and redirection phases

result of the rewrite, the graph G_3 as depicted in Figure 6.28.

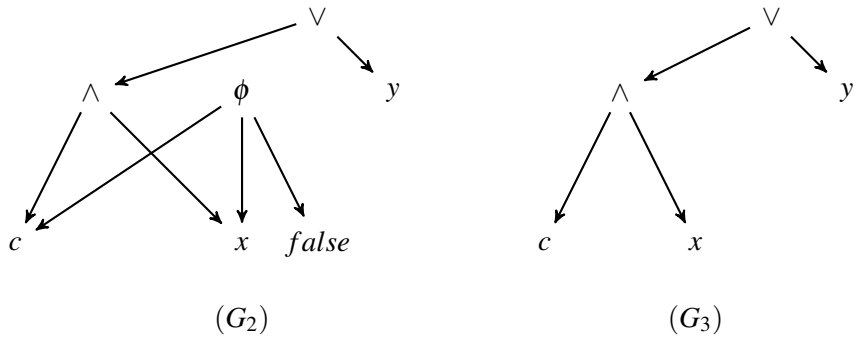

Fig. 6.28 Graph rewriting: Garbage collection phases

6.4.2 Normalizing

Once a shared value-graph is constructed for the SIGNAL program and its generated C program, if the values of an output signal and its corresponding variable in the C program are not already equivalent (they do not point the same node in the shared value-graph), we start to normalize the graph. Given a set of term rewrite rules, the normalizing process works as described in Listing 6.8.

Listing 6.8 Normalizing value-graph

```
Input: G: A shared value-graph.
       R: The set of rewrite rules.
       S: The sharing among graph nodes.
Output: The normalized graph

while (∃s∈S or ∃r∈R that can be applied on G) do
{
   while (∃r∈R that can be applied on G)
   {
   for (n∈G)
      if (r can be applied on n)
         apply the rewrite rule to n
   }
   maximize sharing
}
return G
```

The normalizing algorithm indicates that we apply the rewrite rules to each graph node individually. When there is no more rules that can be applied to the resulting graph, we maximize the shared nodes. The process terminates when there exists no more sharing or

rules that can be applied.

We classify our set of rewrite rules into three basic types: *general simplification rules*, *optimization-specific rules* and *synchronous rules*. In the following, we shall present the rewrite rules of these types, and we assume that all nodes in our shared value-graph are typed. Note that we only write the rewrite rules in form of term rewrite rules, $t_l \rightarrow t_r$.

6.4.2.1 General simplification rules

The general simplification rules contain the rules which are related to the general rules of inference of operators, denoted by the corresponding function symbols in F. In our consideration, the operators used in the primitive *stepwise functions* and in the generated C code are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`), and numerical operators (`+`, `-`, `*`, `/`). When applying these rules, we will replace a subgraph rooted at a node by a smaller subgraph. In consequence of this replacement, they will reduce the number of nodes by eliminating some unnecessary structures.

$$= (t,t) \rightarrow \texttt{true} \tag{6.1}$$

$$\neq (t,t) \rightarrow \texttt{false} \tag{6.2}$$

$$= (t,\texttt{true}) \rightarrow t \tag{6.3}$$

$$\neq (t,\texttt{true}) \rightarrow \neg t \tag{6.4}$$

$$= (t,\texttt{false}) \rightarrow \neg t \tag{6.5}$$

$$\neq (t,\texttt{false}) \rightarrow t \tag{6.6}$$

The first set of general simplification rules simplifies applied numerical and Boolean comparison expressions. In these rules, the term t represents a structure of value computing (e.g., the computation of expression $b = x \neq \texttt{true}$). The rules 6.3, 6.4, 6.5, and 6.6 only apply on the Boolean type. These rules are self explanatory, for instance, with any structure represented by a term t, the expression $t = t$ can always be replaced with the value `true`.

The second set of general simplification rules eliminates unnecessary nodes in the graph that represent the ϕ-functions, where c, c_1 and c_2 are Boolean expressions. For better representation, we divide this set of rules into several subsets as follows.

$$\phi(\texttt{true}, x_1, x_2) \rightarrow x_1 \tag{6.7}$$

$$\phi(\texttt{false}, x_1, x_2) \rightarrow x_2 \tag{6.8}$$

The rules in this set replace a ϕ-function with its left branch if the condition always holds the value `true`. Otherwise, if the condition holds the value `false`, it is replaced with its right branch.

$$\phi(c, \texttt{false}, \texttt{true}) \rightarrow \neg c \tag{6.9}$$

$$\phi(c, \texttt{true}, \texttt{false}) \rightarrow c \tag{6.10}$$

The rules operate on Boolean expressions represented by the branches. When the branches are Boolean constants and hold different values, the ϕ-function can be replaced with the value of the condition c.

$$\phi(c, \texttt{false}, x) \rightarrow \neg c \wedge x \tag{6.11}$$

$$\phi(c, \texttt{true}, x) \rightarrow c \vee x \tag{6.12}$$

$$\phi(c, x, \texttt{false}) \rightarrow c \wedge x \tag{6.13}$$

$$\phi(c, x, \texttt{true}) \rightarrow \neg c \vee x \tag{6.14}$$

The rules operate on Boolean expressions represented by the branches. When one of the branches is Boolean constant, the ϕ-function can be replaced with a Boolean expression of the condition c and the non-constant branch. For instance, when the left branch is a constant and holds the value `true`, then the ϕ-function is replaced with the Boolean expression $c \vee x$.

$$\phi(c, x, x) \rightarrow x \tag{6.15}$$

The rule 6.15 removes the ϕ-function if all of its branches contain the same value. A ϕ-function with only one branch is a special case of this rule. It indicates that there is only one

path to the ϕ-function as happens with branch elimination.

$$\phi(c,\phi(c,x_1,x_2),x_3) \rightarrow \phi(c,x_1,x_3) \tag{6.16}$$
$$\phi(c,x_1,\phi(c,x_2,x_3)) \rightarrow \phi(c,x_1,x_3) \tag{6.17}$$
$$\phi(c,\phi(\neg c,x_1,x_2),x_3) \rightarrow \phi(c,x_2,x_3) \tag{6.18}$$
$$\phi(c,x_1,\phi(\neg c,x_2,x_3)) \rightarrow \phi(c,x_1,x_2) \tag{6.19}$$
$$\phi(c_1,\phi(c_2,x_1,x_2),x_3) \rightarrow \phi(c_1,x_1,x_3) \text{ if } c_1 \Rightarrow c_2 \tag{6.20}$$
$$\phi(c_1,\phi(c_2,x_1,x_2),x_3) \rightarrow \phi(c_1,x_2,x_3) \text{ if } c_1 \Rightarrow \neg c_2 \tag{6.21}$$
$$\phi(c_1 \wedge c_2,\phi(c_1,x_1,x_2),x_3) \rightarrow \phi(c_1 \wedge c_2,x_1,x_3) \tag{6.22}$$
$$\phi(c_1 \wedge c_2,\phi(c_2,x_1,x_2),x_3) \rightarrow \phi(c_1 \wedge c_2,x_1,x_3) \tag{6.23}$$
$$\phi(c_1,x_1,\phi(c_2,x_2,x_3)) \rightarrow \phi(c_1,x_1,x_2) \text{ if } \neg c_1 \Rightarrow c_2 \tag{6.24}$$
$$\phi(c_1,x_1,\phi(c_2,x_2,x_3)) \rightarrow \phi(c_1,x_1,x_3) \text{ if } \neg c_1 \Rightarrow \neg c_2 \tag{6.25}$$
$$\phi(c_1 \vee c_2,x_1,\phi(c_1,x_2,x_3)) \rightarrow \phi(c_1 \vee c_2,x_1,x_3) \tag{6.26}$$
$$\phi(c_1 \vee c_2,x_1,\phi(c_2,x_2,x_3)) \rightarrow \phi(c_1 \vee c_2,x_1,x_3) \tag{6.27}$$

Consider a ϕ-function such that one of its branches is another ϕ-function. The rules 6.16 to 6.27 remove the ϕ-function in the branch if one of the following conditions is satisfied:

- The conditions of the ϕ-functions are the same (as in the rules 6.16 and 6.17).
- The condition of the first ϕ-function is equivalent to the negation of the condition of the second ϕ-function (as in the rules 6.18 and 6.19).
- The condition of the first ϕ-function either implies the condition of the second ϕ-function or its negation (as in the rules 6.20 to 6.23).
- The negation of the condition of the first ϕ-function either implies the condition of the second ϕ-function or its negation (as in the rules 6.24 to 6.27).

The following code segment in C illustrates the use of the above rewrite rules:

```
1 if (c)
2 {
3     a = 0; b = 0; d = a;
4 }
5 else
6 {
7     a = 1; b = 1; d = 0;
```

```
8 }
9 if (a == b)
10    x = d;
11 else
12    x = 1;
13 return x;
```

If we analyze this code segment the return value is 0. In fact, a and b have the same value in both branches of the first "if" statement. Thus in the second "if" statement the condition is always `true`, then x always holds the value of d which is 0. We shall apply the general simplification rules to show that the value-graph of this code segment can be transformed to the value-graph of the value 0. We represent the value-graph in form of linear notation. The value-graph of the computation of x is $\phi(=(a,b),d,0)$. Replacing the definition of a,b and d, and normalizing this graph, we get:

$$
\begin{aligned}
x \mapsto & \phi(=(\phi(c,0,1),\phi(c,0,1)),\phi(c,\phi(c,0,1),0),0) \\
& \phi(\texttt{true},\phi(c,\phi(c,0,1),0),0) && \text{by (6.1)} \\
& \phi(c,\phi(c,0,1),0) && \text{by (6.7)} \\
& \phi(c,0,0) && \text{by (6.16)} \\
& 0 && \text{by (6.15)}
\end{aligned}
$$

6.4.2.2 Optimization-specific rules

Based on the optimizations of the SIGNAL compiler, we have a number of *optimization-specific rules* in a way that reflexes the effects of specific optimizations of the compiler. These rules do not always reduce the graph or make it simpler. One has to know specific optimizations of the compiler when she wants to add them to the set of rewrite rules. In our case, the set of rules for simplifying constant expressions of the SIGNAL compiler is given as follows.

- Specific rules for constant expressions with numerical operators:

$$+(cst_1,cst_2) \rightarrow cst, \text{ where } cst = cst_1 + cst_2 \tag{6.28}$$

$$*(cst_1,cst_2) \rightarrow cst, \text{ where } cst = cst_1 * cst_2 \tag{6.29}$$

$$-(cst_1,cst_2) \rightarrow cst, \text{ where } cst = cst_1 - cst_2 \tag{6.30}$$

$$/(cst_1,cst_2) \rightarrow cst, \text{ where } cst = cst_1 / cst_2 \tag{6.31}$$

- Specific rules for constant expressions with usual logic operators:

$$\neg\texttt{false} \rightarrow \texttt{true} \tag{6.32}$$

$$\neg\texttt{true} \rightarrow \texttt{false} \tag{6.33}$$

$$\wedge(cst_1, cst_2) \rightarrow cst, \text{ where } cst = cst_1 \wedge cst_2 \tag{6.34}$$

$$\vee(cst_1, cst_2) \rightarrow cst, \text{ where } cst = cst_1 \vee cst_2 \tag{6.35}$$

- Specific rules for constant expressions with numerical comparison functions:

$$\square(cst_1, cst_2) \rightarrow cst \tag{6.36}$$

where $\square$ = <, >, =, <=, >=, /=, and the Boolean value cst is the evaluation of the constant expression $\square(cst_1, cst_2)$ which can hold either the value `false` or `true`.

We also may add a number of rewrite rules that are derived from the list of *rules of inference* for propositional logic. For example, we have a group of laws for rewriting formulas with `and` operator, such as:

$$\wedge(x, \texttt{false}) \rightarrow \texttt{false}$$

$$\wedge(x, \texttt{true}) \rightarrow x$$

$$\wedge(x, \Rightarrow(x, y)) \rightarrow x \wedge y$$

We consider the following SIGNAL program and its generated C code, the input signal x is present when the other Boolean input signal cx holds the value `true`.

```
1 /* Signal equation */
2 | x ^= when cx
3 /* Generated C code */
4 if (C_cx)
5 {
6    if (cx)
7    {
8       if (!r_P_x(&x)) return FALSE;
9    }
10 }
```

The computation of x is represented by $x = \phi(\hat{x}, \widetilde{x}, \perp)$, where $\hat{x} \Leftrightarrow \widehat{cx} \wedge \widetilde{cx}$. In the generated C code, the value of x is read only when the condition $C\_cx \wedge cx$ is `true`. That is represented by $x = \phi(C\_cx, \phi(cx, \widetilde{x}, \perp), \perp)$. This observation makes us add the following rewrite rule

into the systems to mirror the above rewriting of the SIGNAL compiler.

$$\phi(c_1, \phi(c_2, x_1, x_2), x_2) \rightarrow \phi(c_1 \wedge c_2, x_1, x_2) \tag{6.37}$$

6.4.2.3 Synchronous rules

In addition to the general and optimization-specific rules, we also have a number of rewrite rules that are derived from the semantics of the code generation mechanism of the SIGNAL compiler. To illustrate why the synchronous rules need to be added in our validator, we consider the SIGNAL program in Listing 6.9 and the corresponding generated C code in Listing 6.10. The shared value-graph of the SIGNAL program and its generated C code is given in Figure 6.29.

Listing 6.9 MasterClk in SIGNAL

```
process MasterClk=
(? integer x;
 ! integer z)
 (| z := x default 0
  | pz := z$1 init 0
  | x ^= when (pz <= 1)
  |)
where integer pz
end;
```

Listing 6.10 Generated C code of MasterClk

```
EXTERN logical MasterClk_step()
{
   C_x = z <= 1;
   if (C_x)
   {
      if (!r_MasterClk_x(&x)) return FALSE;
   }
   if (C_x) z = x; else z = 0;
   w_MasterClk_z(z);
   MasterClk_step_finalize();
   return TRUE;
}
```

In this example, the fastest clock is $\hat{z}$, called the *master clock*. All other clocks are expressed as calculus expression of the master clock (the clock $\hat{x}$ is an under-sampling of $\hat{z}$ according to

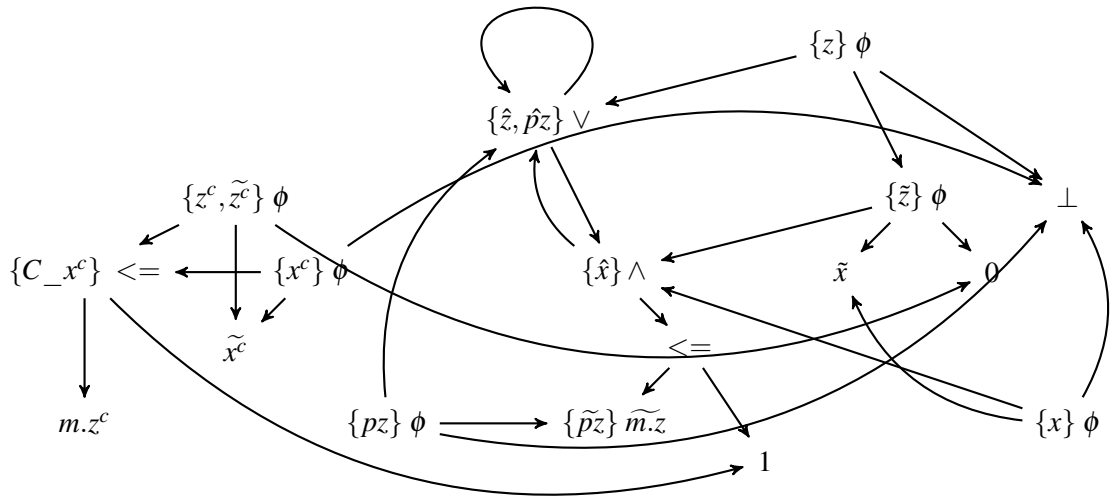

Fig. 6.29 The shared value-graph of MasterClk and MasterClk_step

the values of Boolean expression $pz <= 1$). Such programs are referred to as endochronous, and can be executed in a deterministic way.

Consider the generated C code, we observe that the value of the variable z is always updated. It holds the value of x if C_x is `true`, otherwise it is 0. Hence, we have the following rule that mirrors the above situation.

$$x^c \mapsto \phi(\texttt{true}, \widetilde{x^c}, \perp) \rightarrow x \mapsto \phi(\texttt{true}, \tilde{x}, \perp) \tag{6.38}$$

We write $x \mapsto \phi(\texttt{true}, \tilde{x}, \perp)$ to denote that x points to the subgraph rooted at the node labeled by ϕ-function. The rule 6.38 indicates that if a variable in the generated C code is always updated, then we require that the corresponding signal in the source program is present at every instant, meaning that the signal never holds the absent value. In consequence of this rewrite rule, the signal x and its value when it is present $\tilde{x}$ (resp. the variable x^c and its updated value $\widetilde{x^c}$ in the generated C code) point to the same node in the shared value-graph. Every reference to x and $\tilde{x}$ (resp. x^c and $\widetilde{x^c}$) point to the same node.

For example, consider the value-graph in Figure 6.29, we rewrite the subgraph representing the clock $\hat{z}$ and $\hat{pz}$ into a single node labeled by the value `true`. Then, we apply the rule 6.7 on the resulting graph, and we obtain the reduced graph in Figure 6.30.

A second synchronous rule mirrors the semantics of the *delay* operator. For instance, we consider the equation $pz := z\$1$ `init` 0 in Listing 6.9. We use the variable $\widetilde{m.z}$ to capture the last value of the signal z. In the generated C program, the last value of the variable z^c is

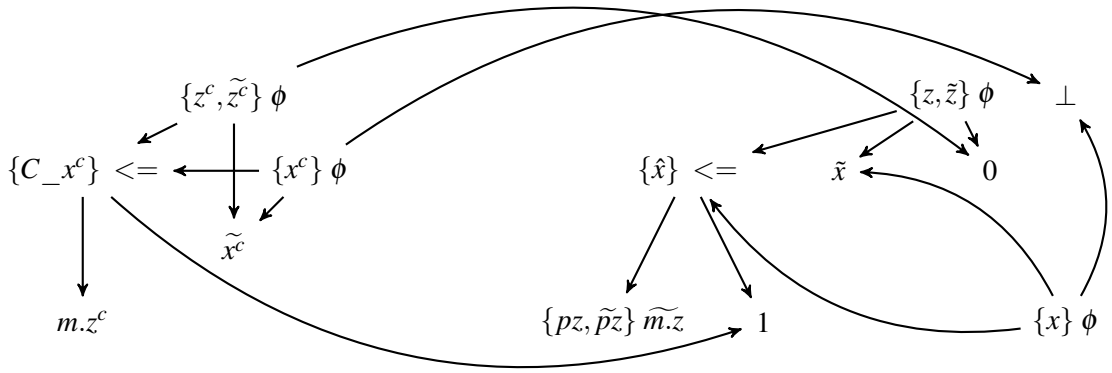

Fig. 6.30 The resulting graph of MasterClk and MasterClk_step by applying the rule 6.38

denoted by $m.z^c$. We require that the last values of a signal and the corresponding variable in the generated C code are the same. That means $\widetilde{m.z} = m.z^c$.

$$m.x^c \mapsto G_1 + \widetilde{m.x} \mapsto G_2 \rightarrow m.x^c, \widetilde{m.x} \mapsto G_1 \tag{6.39}$$

The rule 6.39 indicates that for any signal x which is involved in a *delay* operator, and its corresponding variable in the generated C program, then it is required that the last values of x and x^c are the same. Therefore, every reference to $m.x^c$ and $\widetilde{m.x}$ points to the same node.

Consider the value-graph in Figure 6.30, $m.z^c$ and $\widetilde{m.z}$ point to the same node by rule 6.39. Then, C_x and $\hat{x}$ are represented by the same subgraph. And any reference to C_x and $\hat{x}$ points to the same node in the graph as depicted in Figure 6.31.

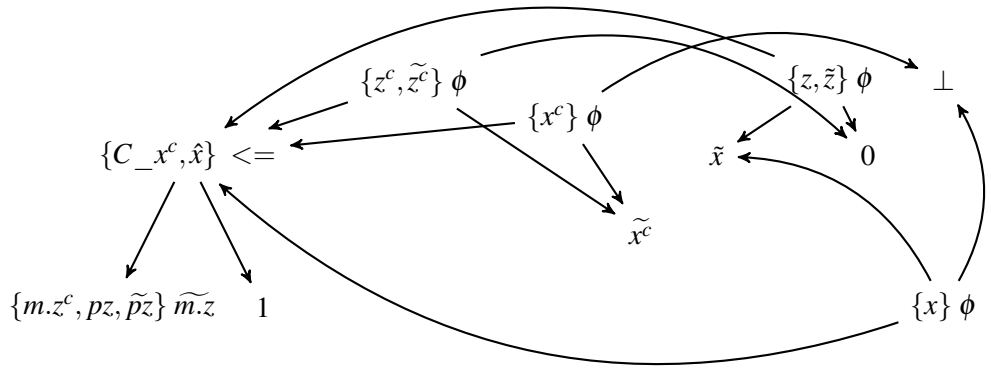

Fig. 6.31 The resulting graph of MasterClk and MasterClk_step by applying the rule 6.39

Finally, we add rules that mirror the relation between input signals and their corresponding variables in the generated C code. First, for any input signal x and the corresponding variable x^c in the generated C code, if x is present, then the value of x which is read from the environment and the value of the variable x^c after the reading statement must be equivalent. That means $\widetilde{x^c}$ and $\tilde{x}$ are represented by the same subgraph in the graph. Second, if the clock of x is also read from the environment as a parameter, then the clock of the input signal x is equivalent to the condition in which the variable x^c is updated. It means that we represent $\hat{x}$ and C_x^c by the same subgraph.

$$\widetilde{x^c} \mapsto G_1 + \tilde{x} \mapsto G_2 \rightarrow \widetilde{x^c}, \tilde{x} \mapsto G_1 \tag{6.40}$$

$$C\_x^c \mapsto G_1 + \hat{x} \mapsto G_2 \rightarrow C\_x^c, \hat{x} \mapsto G_1 \tag{6.41}$$

Consequently, every reference to $\hat{x}$ and C_x^c (resp. $\tilde{x}$ and $\widetilde{x^c}$) points to the same node. Considering the value-graph of the program in Listing 6.10 and its generated C code, by rule 6.40 we obtain the final normalized graph that is depicted in Figure 6.32. We can observe that the value of the signal z and the corresponding variable z^c are represented by the same subgraph. Therefore, we can safely conclude that the value of output signal z and the corresponding variable z^c in the generated C code are equivalent.

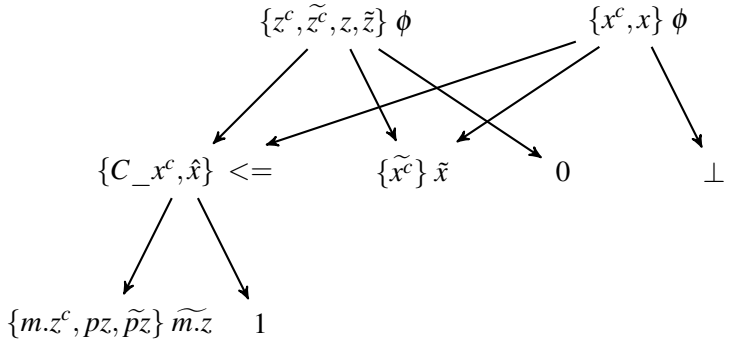

Fig. 6.32 The final normalized graph of MasterClk and MasterClk_step

6.4.3 Implementation

Given a SIGNAL program A, with an unverified compiler Cp, we consider the following derived compiler that is followed by a validator. The validator checks that for any output signal x in the source program A and the corresponding variable x^c in the C program, they have the same values ($\tilde{x} = \widetilde{x^c}$). In other words, $\tilde{x}$ and $\widetilde{x^c}$ point to the same node in the shared value-graph. We denote this fact by $C \sqsubseteq_{val} A$.

```
1 if (Cp(A) is Error) return Error;
2 else
3 {
4    if (C ⊑val A) return C;
5    else return Error;
6 }
```

The main components of the validator are depicted in Figure 6.33. The validator works as follows. First, it constructs a shared value-graph that represents the computation of all signals and variables in both programs. The value-graph can be considered as a generalization of symbolic evaluation. The shared value-graph is transformed by applying some graph rewrite rules, this process is called normalization. The set of rewrite rules reflexes the general rules of inference of operators, or the optimizations of the compiler. For instance, consider the 3-node subgraph representing the expression $1 > 0$, the normalization will transform that graph into a single node subgraph representing the value `true`, as it reflexes the constant folding.

Finally, it compares the values of the output signals and the corresponding variables in the C program. For every pair of output signal and its corresponding variable, the validator checks whether they point to the same node in the shared graph. Therefore, in the best case, when semantics has been preserved, this checking has constant time complexity $\mathcal{O}(1)$. In fact, it is always expected that most transformations and optimizations are semantics-preserving, thus the best-case complexity is important.

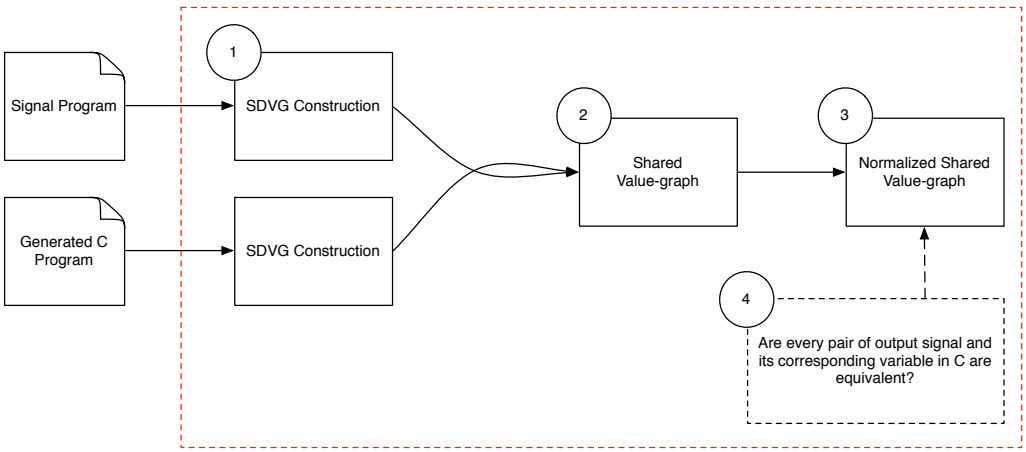

Fig. 6.33 A bird's-eye view of the SDVG translation validation

Let us illustrate the above steps on the program `DEC` in Listing 3.3 and its generated C code `DEC_step()` in Listing 6.11.

In the first step, we shall compute the shared value-graph for both programs to represent the computation of all signals and their corresponding variables. This graph is depicted in Figure 6.34.

Listing 6.11 Generated C code of DEC

```
1 EXTERN logical DEC_step()
2 {
3    C_FB = N <= 1;
4    if (C_FB)
5    {
6       if (!r_DEC_FB(&FB)) return FALSE;
7    }
8    if (C_FB) N = FB; else N = N - 1;
9    w_DEC_N(N);
10   DEC_step_finalize();
11   return TRUE;
12 }
```

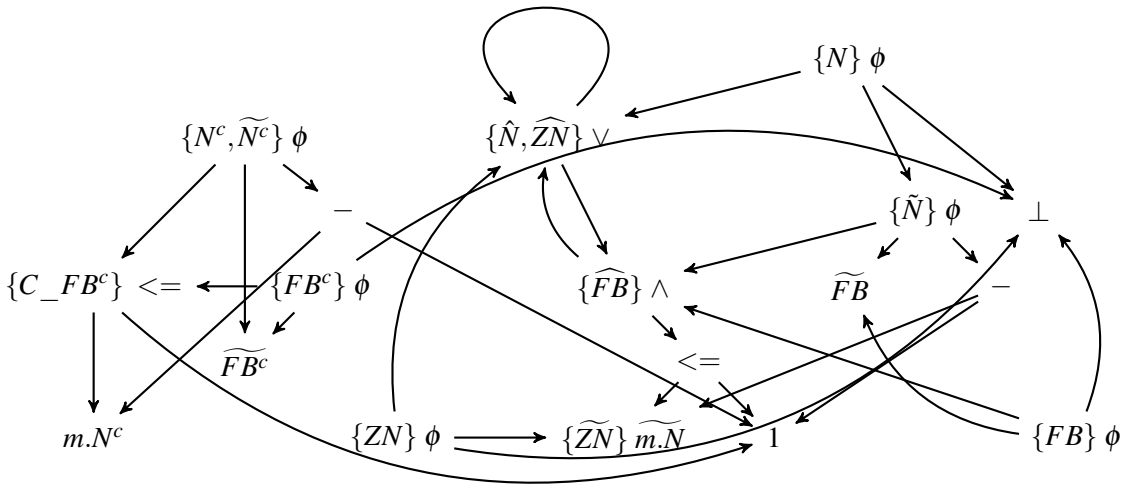

Fig. 6.34 The shared value-graph of DEC and DEC_step

Note that in the C program, the variable N^c ("c" is added as superscript for the C program variables, to distinguish them from the signals in the SIGNAL program) is always updated (line (8)). In lines (3) and (8), the references to the variable N^c are the references to the last value of N^c denoted by $m.N^c$. The variable FB^c which corresponds to the input signal FB is updated only when the variable C_FB^c is `true`.

In the second step, we shall normalize the above initial graph. Below is a potential normalization scenario, meaning that there might have more than one normalization scenario, and the validator can choose one of those. For example, given a set of rules that can be applied, the validator can apply these rules with different order. Figure 6.35 depicts the intermediate resulting graph of this normalization scenario. And Figure 6.36 is the final normalized graph from the initial graph when we cannot perform any more normalization.

- The clock of the output signal N is a master clock which is indicated in the generated C by the variable N^c being always updated. By rule 6.38, the node $\{\hat{N}, \widehat{ZN}\} \vee$ is rewritten into `true`.
- By rule $\wedge(\texttt{true}, x) \rightarrow x$, the node $\{\widehat{FB}\} \wedge$ is rewritten into $\{\widehat{FB}\}$ $<=$.
- The node ϕ-function representing the computation of N is removed and N points to the node $\{\tilde{N}\}$ ϕ by rule 6.7.
- The node ϕ-function representing the computation of ZN is removed and ZN points to the node $\{\widetilde{ZN}\}$ $\widetilde{m.N}$ by rule 6.7.
- The nodes $\widetilde{FB^c}$ and $\widetilde{FB}$ are rewritten into a single node $\{\widetilde{FB}\}$ $\widetilde{FB^c}$ by rule 6.40. And all references to them are replaced by the references to $\{\widetilde{FB}\}$ $\widetilde{FB^c}$.
- The nodes $m.N^c$ and $\widetilde{m.N}$ are rewritten into a single node $\{\widetilde{m.N}\}$ $m.N^c$ by rule 6.39. And all references to them are replaced by the references to $\{\widetilde{m.N}\}$ $m.N^c$.

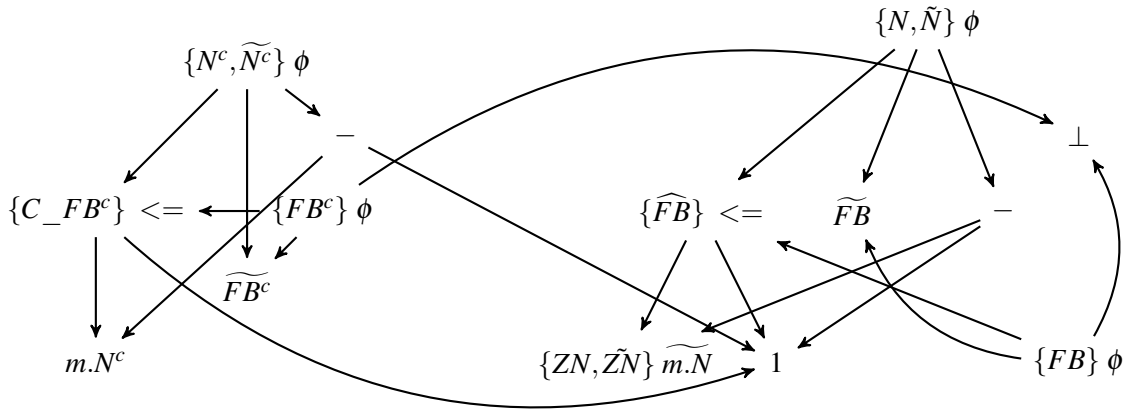

Fig. 6.35 The resulting value-graph of DEC and DEC_step

In the final step, we check that the value of the output signal and its corresponding variable in the generated code merge into a single node. In this example, we can safely

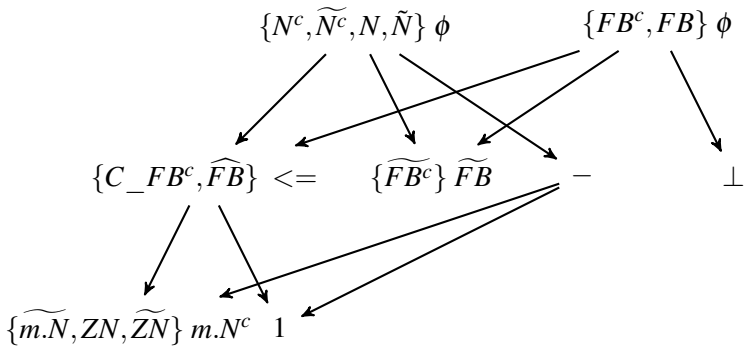

Fig. 6.36 The final normalized graph of DEC and DEC_step

conclude that the output signal N and its corresponding variable N^c is equivalent since they point to the same node in the final normalized graph.

6.5 Discussion

There is a wide range of works for value-graph representations of expression evaluations in a program. For example, in [137], Weise et al. present a nice summary of the various types of value-graph. In our context, the value-graph is used to represent the computation of variables in both source program and its generated C code and identical structures are shared. We believe that this representation will reduce the required storage and make the normalizing process more efficient. Another remark is that the calculation of clocks as well as the special value, the absent value, are also represented in the shared graph.

Another related work which adopts the translation validation approach in verification of optimizations, Tristan et al. [135], recently proposed a framework for translation validation of LLVM optimizer. For a function and its optimized counterpart, they compute a shared value-graph. The graph is *normalized* (the graph is reduced). After the normalizing, if the outputs of two functions are represented by the same sub-graph, they can safely conclude that both functions are equivalent.

On the other hand, Tate et al. [132] proposed a framework for translation validation. Given a function in the input program and the corresponding optimized version of the function in the output program, they compute two value-graphs to represent the computations of the variables. Then they transform the graph by adding equivalent terms through a process called *equality saturation*. After the saturation, if both value-graphs are the same, they can conclude that the return value of two given functions are the same. However, for the translation validation purpose, our normalization process is more efficient and scalable since we

can add some rewrite rules into the validator that reflects what a typical compiler intends to do (e.g., a compiler will do the constant folding optimization, then we can add the rewrite rule for constant expressions such as three nodes subgraph $1+2$ is replaced by a single node 3).

The present chapter provides a proof of the preservation of value-equivalence and applies this approach to the synchronous data-flow compiler SIGNAL. With the simplicity of the graph normalization, we believe that translation validation of synchronous data-flow value-graph for the industrial compiler SIGNAL is feasible and efficient. Moreover, the normalization process can always be extended by adding new rewrite rules. That makes the translation validation of SDVG scalable and flexible.

The proof of the preservation of value-equivalence might be extended to the whole compilation process of the SIGNAL compiler. To realize that, we shall prove that for every signal in the source program, if it is present then the corresponding variable in the generated C code is updated. Otherwise, if the signal is absent then the corresponding variable is never updated. In other words, for every clock associated with a signal and the corresponding Boolean value in the generated C code, thay are represented by the same subgraph. However, we have claimed in Chapter 1 that it is better to separate the concerns and proving for each phase the preservation of different kinds of semantics.

CHAPTER 7

CONCLUSION

7.1 Summary of the contribution

Our aim in this dissertation was to present an approach to compiler verification that adopts translation validation to build a formally verified compiler verifier within the existing SIGNAL toolset as described in Figure 1.5. By further decomposing translation validation following the successive compilation steps of the SIGNAL compiler, we were able to narrow down the problem to simpler proofs on simpler properties that observational equivalence. Moreover, the validator is smaller, modular and insensitive to minor updates of the compiler. In consequence, this works experiments with a solution that is in total significantly lighter in terms of efforts than a formal compiler verification, while providing, if not the same, a tangible level of assurance with respect to DO-330 requirements.

To realize this aim, we did not prove that all source programs and their generated code have the same semantics. Instead, we separated the proof into three smaller, independent, sub-problems: the proof of clock semantic preservation, the proof of data dependency preservation, and the proof of value-equivalence preservation of variables. These smaller proofs apply on different, successive, phases of the compilation process.

We have designed three special-purpose validators for each phase of the compilation process to carry out the proof of correctness. The validators benefit from the advantages of the translation validation approach and satisfy correctness requirements. The design of the validators is novel and constitutes the primary contribution of this work.

When solving the preservation of clock semantics problem, we defined a common semantic representation, called *clock model*, that is inspired by the interval-Boolean abstraction to represent clock semantics. With the efficiency of SMT solving, we found that representing the clock semantics with clock models is suitable and profitable compared to the

approach based on model checking. The validator can deal with large programs and avoid the state-space explosion problem.

We define the concept of SDDGs as the common semantic framework in the translation validation to prove the preservation of data dependencies. The dependencies are dynamically represented by conditioning the dependencies with Boolean expressions. SDDGs differ from GCDs in the labels of the edges in the graphs. A refinement relation between two graphs expresses the preservation of the dependencies, whose existence can be checked, again, by using an SMT solver. In addition, a synchronous data-flow dependency graph in which the labels between nodes are encoded in the interval-Boolean abstraction, is used to make a more precise deadlock detection for the SIGNAL compiler.

For an output program of the *static scheduling* phase and its generated C program, we construct a shared value-graph to represent the computation of all signals and variables in both programs. This shared graph is *normalized* using a set of rewriting rules. An output signal x and its corresponding variable are checked equivalent iff they are represented by the same sub-graph. Symbolic evaluation and graph rewriting turn out to be valuable techniques to build validators. We believe that the use of symbolic evaluation and graph rewriting to prove the preservation of value-equivalence is novel. Moreover, normalization makes proofs scalable and flexible.

Finally, we have shown that it is possible to construct a validator to prove the correctness of the whole compilation process of the SIGNAL compiler. This validator can be considered as the modular composition of special-purpose validators. It ensures that the clock semantics, data dependencies among signals, and value-equivalence of variables are preserved when the compiler compiles a SIGNAL program into the corresponding generated C program. Should a new compilation module or phase be added, such as desynchronization, one could add another dedicated validator to prove its specific transformation correct.

The work on clock semantics has been published in the proceedings of the 9th International Conference on Integrated Formal Methods [106] and presented in June 2012 in Pisa, Italy. The work on clock semantics of the code generation phase was accepted for presentation at IEEE International High-Level Design, Validation and Test Workshop HLDVT'12 in California, USA [107]. The combined work on clock semantics and data dependencies has been published in Frontiers of Computer Science with special issue on Synchronous Programming, Springer journal in 2013 [108]. The deadlock detection technique in Section 5.4 has been published in the proceedings of the 2014 Electronic System Level Synthesis Conference, ESLsyn 2014 [109] and presented in June 2014 in San Francisco, USA. The translation validation of synchronous data-flow value-graph is being prepared for some

international conference and peer-review journal.

7.2 Future work

There is a number of possible directions for extending the proof of correctness of synchronous compiler. This work showed the use of SMT solver to prove the preservation of clock semantics and data dependencies. One possible next work is to fully implement the validators and the sets of experiments which demonstrate the capability of dealing with very large programs.

Considering the last phase of the compilation process, the executable code generation from SIGNAL to sequential C code, we provide a proof of correctness of the SIGNAL compiler for the preservation of value-equivalence. A possibility is to extend this proof to use with the other code generation schemes including cluster code with static and dynamic scheduling, modular code, and distributed code. One path forward is the combination of the work in Chapter 5 and Chapter 6. That means that we use synchronous data-flow dependency graphs and synchronous data-flow value-graphs as a common semantic framework to represent the semantics of the generated code. The formalization of the notion of "correct transformation" is defined as the refinements between two synchronous data-flow dependency graphs and in a shared value-graph as described in previous chapters.

Another possibility is that we use a SMT solver to reason on the rewriting rules. For example, we recall the following rules:

$$\phi(c_1,\phi(c_2,x_1,x_2),x_3) \rightarrow \phi(c_1,x_1,x_3) \text{ if } c_1 \Rightarrow c_2$$
$$\phi(c_1,\phi(c_2,x_1,x_2),x_3) \rightarrow \phi(c_1,x_2,x_3) \text{ if } c_1 \Rightarrow \neg c_2$$

To apply these rules on a shared value-graph to reduce the nested ϕ-functions (e.g., from $\phi(c_1,\phi(c_2,x_1,x_2),x_3)$ to $\phi(c_1,x_1,x_3)$), we have to check the validity of first-order logic formulas, for instance, we check that $\models (c_1 \Rightarrow c_2)$ and $\models c_1 \Rightarrow \neg c_2$. We consider the use of SMT to solve the validity of the conditions as in the above rewrite rules to normalize value-graphs.

The SME environment based on Model-Driven Engineering (MDE) technologies in the ECLIPSE environment is a front-end of the POLYCHRONY toolset that relies on the Eclipse Modeling Framework (EMF). It allows developers to write SIGNAL programs as graphical models based on the TOPCASED modeling facilities [134]. It would be interesting to construct a validator for the translation from graphical models into SIGNAL programs. It would

also be profitable to extend this validator to other translators from graphical models such as SIMULINK tool of MathWorks [129] that translate diagrams into C/C^{++} programs.

REFERENCES

[1] ISO 14882. Standard for programming language c++. https://isocpp.org/files/papers/N3690.pdf, 2013.

[2] ACE. Ace supertest suite. http://www.ace.nl/compiler/supertest.html, 2013.

[3] W. Ackerman. Solvable cases of the decision problem. *Study in Logic and the Foundations of Mathematics. North-Holland, Amsterdam*, 1954.

[4] Ada. The programming language ada reference manual. *New York: Springer Verlag LNCS 155*, 1983.

[5] G. Alefeld and J. Hertzberger. Introduction to interval computation. *Academic Press. New York*, 1983.

[6] F.E. Allen. Control flow analysis. *In proceedings of a Symposium on Compiler Optimization, SIGPLAN*, pages 1–19, 1970.

[7] T. Amagbegnon, L. Besnard, and P. Le Guernic. Arborescent canonical form of boolean expressions. *INRIA Report n.2290*, 1994.

[8] Arvind and K.P. Gostelow. Some relationships between asynchronous interpreters of data-flow language. *North-Holland*, 1978.

[9] Astrée. The static program analyzer. http://www.astree.ens.fr/, 2014.

[10] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of signal programs. *In Proceedings of the 29th Hawaii International Conference on System Sciences, IEEE Computer Society Press*, 1:656–665, 1996.

[11] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: A representation supporting control, data, and demand driven interpretation of imperative languages. *In Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271, 1990.

[12] H.P. Barendregt, M.C. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards an intermediate language based on graph rewriting. *These Proceedings van den Broek, P.M and G.F van der Hoeven*, 1987.

[13] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Transactions on Automatic Control*, 35(5):535–546, 1990.

[14] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[15] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *In Proceedings of the IEEE*, 91(1), 2003.

[16] G. Berry. Real-time programming: Special purpose or general purpose languages. *In IFIP World Computer Congress, San Francisco*, 1989.

[17] G. Berry. The foundations of esterel. *In Proof, Language and Interaction: Essay in Honor of Robin Milner, MIT Press*, 2000.

[18] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems, an introduction to esterel. *In Programming of Future Generation Computers, Elsevier Science Publisher B.V. North Holland*, 1988.

[19] L. Besnard, T. Gautier, P. Le Guernic, and J-P. Talpin. Compilation of polychronous data-flow equations. *In Synthesis of Embedded Software Springer*, pages 1–40, 2010.

[20] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. *In Int. Conf. On Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, 1999.

[21] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, 2009.

[22] D. Biernacki, J-L. Colaco, G. Hamon, and M. Pouzet. Clock-directed modular code generation of synchronous data-flow languages. *In ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2008.

[23] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *In The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, T. Mogensen and D.A. Schmidt and I.H. Sudborough (Editors)*, 2566 of Lecture Notes in Computer Science:85–108, 2002.

[24] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. *In PLDI 2003 — ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation*, page 196–207, 2003.

[25] S. Blazy. Which c semantics to embed in the front-end of a formally verified compiler? *Tools and Techniques for Verification of System Infrastructure, TTVSI*, 2008.

[26] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a c compiler front-end. *14th Symposium on Formal Methods (FM'06), Lecture Notes in Computer Science (LNCS 4085), Springer*, pages 460–475, 2006.

[27] S. Blazy, B. Robillard, and A.W. Appel. Formal verification of coalescing graph-coloring register allocation. *19th European Symposium On Programming (ESOP 2010), Lecture Notes in Computer Science (LNCS 6012), Springer*, pages 145–164, 2010.

[28] E. Borger, E. Gradel, and Y. Gurevich. The classical decision problem. *Spinger-Verlag*, 1996.

[29] M. Le Borgne, A. Benveniste, and P. Le Guernic. Dynamical systems over galois fields and control problems. *In Proceedings of 33th IEEE on Decision and Control*, 3:1505–1509, 1991.

[30] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.P. Talpin. Integrating system descriptions by clocked guarded actions. *In FDL, IEEE*, pages 1–8, 2011.

[31] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.P. Talpin. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. *Design Automation for Embedded Systems*, pages 1–35, 2012.

[32] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[33] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the hol theorem prover. *Technical Report 265, Computer Laboratory, University of Cambridge*, 1992.

[34] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[35] L.M. Chirica and D.F. Martin. Towards compiler implementation correctness proofs. *ACM TOPLAS*, 1986.

[36] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *In Int. Conf. On Formal Methods in System Design*, 19:7–34, 2001.

[37] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *In Proceedings of the IBM Workshop on Logics of Programs, Springer-Verlag, Berlin*, 131:52–71, 1981.

[38] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finie-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[39] E.M. Clarke, O. Grumberg, and D.A. Peled. Automatic verification of finie-state concurrent systems using temporal logic specifications. *Model Checking, The MIT Press, Cambridge, MA*, 2000.

[40] CompCert. Compcert c verified compiler. http://compcert.inria.fr/partners.html, 2014.

[41] K. Cooper and L. Torczon. *Engineering a Compiler, Second Edition*. Morgan Kaufmann, 2011.

[42] Coq-Inria. Coq proof assistant. http://coq.inria.fr/, 2014.

[43] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992.

[44] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[45] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6:69–95, 1999.

[46] M.A. Dave. Compiler verification: A bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

[47] J.B. Dennis. First version of a data flow procedure language. *Revised Comp. Struc. Group Memo 93 MIT LCS*, 1975.

[48] J.B. Dennis, J.B. Fosseen, and J.P. Linderman. Data flow schemas. *In A. Ershov and V.A. Nepomniaschy, editors, International Symposium on Theoretical Programming, Lecture Notes in Computer Science*, 5:187–216, 1978.

[49] RTCA DO-178. Software considerations in airborne systems and equipment certification. *RTCA and EUROCAE*, 2011.

[50] RTCA DO-333. Formal methods supplement to do-178c and do-278a. *RTCA and EUROCAE*, 2011.

[51] B. Dutertre and L. de Moura. Yices smt solver. http://yices.csl.ri.com, 2009.

[52] B. Dutertre, M. Le Borgne, and H. Marchand. Sigali: un système de calcul formel pour la vérification de programmes signal. *Manuel d'Utilisation. Note Technique, Non Publiée*, 1998.

[53] H. Ehrig, G. Engels, H-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.2: Applications, Languages and Tools*. World Scientific, 1999.

[54] H. Ehrig, H-J. Kreowski, U. Montanari, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific, 1999.

[55] D. Brown et al. Guidance for using formal methods in a certification context. *Proc. Embedded Real-time Systems and Software*, 2010.

[56] J. Souyris et al. Formal verification of avionics software products. *In Proc. Formal Methods, Springer*, 2009.

[57] P. Feautrier, A. Gamatié, and L. Gonnord. Enhancing the compilation of synchronous data-flow programs with combined numerical-boolean abstraction. *In CSI Journal of Computing*, 1(4):86–99, 2012.

[58] J.C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), 1990.

[59] R.B. França, S. Blazy, D. Favre-Félix, X. Leroy, M. Pantel, and J. Souyris. Formally verified optimizing compilation in acg-based flight control software. *Embedded Real Time Software and Systems, ERTS2*, 2012.

[60] A. Gamatié. *Designing Embedded Systems with the Signal Programming Language*. Springer, 2010.

[61] A. Gamatié and L. Gonnord. Static analysis of synchronous programs in signal for efficient design of multi-clocked embedded systems. *In ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems - LCTES'2011. Chicago, IL, USA*, 2011.

[62] A. Gamatié, T. Gautier, P. Le Guernic, and J-P. Talpin. Polychronous design of embedded real-time applications. *ACM Transactions on Software Engineering and Methodology*, 2005.

[63] A. Gamatié, T. Gautier, and P. Le Guernic. Towards static analysis of signal programs using interval techniques. *In Synchronous Languages, Applications, and Programming (SLAP'06)*, 2006.

[64] T. Gautier and P. Le Guernic. Code generation in the sacres project. *In Towards System Safety, Proceedings of the Safety-critical Systems Symposium*, pages 127–149, 1999.

[65] T. Gautier, P. Le Guernic, and L. Besnard. Signal, a declarative language for synchronous programming of real-time systems. *In Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture, LNCS 274*, 1990.

[66] GeneAuto. Geneauto project. www.geneauto.org, 2014.

[67] L. George and A.W. Appel. Iterated register coalescing. *In TOPLAS*, 18(3):300–324, 1996.

[68] L. Gonnord and N. Halbwachs. Abstract acceleration to improve precision on linear relation analysis. *Research Report, Verimag*, 2010.

[69] M.J.C. Gordon and T.F. Melham. Introduction to hol: A theorem proving environment for higher order logic. *Cambridge University Press*, 1993.

[70] P. Le Guernic and T. Gautier. Advanced topics in data-flow computing, chapter data-flow to von neumann: the signal approach. *Prentice-Hall*, pages 413–438, 1991.

[71] P. Le Guernic, M. Le Borgne, T .Gautier, and C. Le Maire. Programming real-time applications with signal. another look at real-time programming. *Proceedings of the IEEE Special Issue*, 1991.

[72] P. Le Guernic, J-P. Talpin, and J-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, 2003.

[73] N. Halbwachs. A synchronous language at work: the story of lustre. *In 3th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'05)*, 2005.

[74] D. Harel and A. Pnueli. On the development of reactive systems. *In Logic and Models of Concurrent Systems, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, Springer Verlag*, 1985.

[75] A.P. Heffter and M. Gawkoski. Towards proof generating compilers. *Proceedings of COCV*, 2004.

[76] T.A. Henzinger and J. Sifakis. The embedded systems design challenge. *In Formal Method (FM'06), LNCS*, 4085:1–15, 1985.

[77] Inria/Espresso. Polychrony toolset. http://www.irisa.fr/espresso/Polychrony, 2013.

[78] Isabelle. Isabelle proof assistant. http://www.cl.cam.ac.uk/research/hvg/Isabelle/, 2013.

[79] N. Izerrouken, M. Pantel, and X. Thirioux. Machine checked sequencer for critical embedded code generator. *International Conference on Formal Engineering Methods (ICFEM 2009)*, pages 521–540, 2009.

[80] N. Izerrouken, O.S. Yan Kai, M. Pantel, and X. Thirioux. Use of formal methods for building qualified code generator for safer automotive systems. *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness and Safety, ACM*, pages 53–56, 2010.

[81] D. Jackson. A direct path to dependable software. *Communications of the ACM*, pages 52(4):78–88, 2009.

[82] D.B Johnson. Finding all the elementary circuits of a directed graph. *In SIAM J. Comput*, 1(4), 2012.

[83] B. Jose, A. Gamatié, J. Ouy, and S. Shukla. Smt based false causal loop detection during code synthesis from polychronous specifications. *In 9th ACM-IEEE International Conference on Formal Methods and Models for Codesign, IEEE*, 2011.

[84] G. Kahn. The semantics of a simple language for parallel programming. *In J.L. Rosenfeld, editor, Information Processing 74*, pages 471–475, 1974.

[85] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine and compiler. *Technical Report 0400001T.1, National ICT Australia. To appear in ACM TOPLAS*, 2004.

[86] O. Kouchnarenko and S. Pinchinat. Intensional approaches for symbolic methods. *In Electronic Notes in Theoretical Computer Science*, 1998.

[87] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. *In 8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[88] G. Lalire, M. Argoud, and B. Jeannet. Interproc: An inter-procedural analyzer for imperative languages. *http://pop- art.inrialpes. fr/people/bjeannet/bjeannet- forge/interproc*, 2009.

[89] C++ Programming Language. Bitwise operators and bit manipulation. http://www.cplusplus.com/doc/tutorial/operators/, 2014.

[90] E. Ledinot and D. Pariente. Formal methods and compliance to the do-178c/ed12c standard in aeronautics. *Static Analysis of Software*, 2012.

[91] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a c0 compiler. *In Proc. Conf. on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany*, 2005.

[92] X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. *In 33rd Symposium Principles of Programming Languages*, pages 42–54, 2006.

[93] X. Leroy. A formally verified compiler back-end. *In Journal of Automated Reasoning Manuscript*, pages 43(4):363–446, 2009.

[94] X. Leroy and S. Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[95] R. Leviathan and A. Pnueli. Validating software pipelining optimizations. *In Int. Conf. On Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2002)*, pages 280–287, 2006.

[96] Inmos Ltd. The occam programming manual. *Englewood Cliffs, NJ: Prentice-Hall*, 1984.

[97] O. Maffeïs and P. Le Guernic. Distributed implementation of signal: Scheduling and graph clustering. *In 3rd International School and Symposium on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS*, 863:547–566, 1994.

[98] H. Marchand, E. Rutten, M. Le Borgne, and M. Samaan. Formal verification of signal programs: Application to a power transformer station controller. *In Science of Computer Programming*, 41(1):85–104, 2001.

[99] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expression. *Mathematical Aspects of Computer Science*, 1967.

[100] R. Milner. Operational and algebraic semantics of concurrent processes. *Research Report ECS-LFCS-88-46, Lab. for Foundations of Computer Science, Edinburgh*, 1998.

[101] R. Milner and R.W. Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence 7, Edinburgh University Press*, 1972.

[102] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE Software, focus: Safety-Critical Software*, 2013.

[103] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla. Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework. *In Electronic System Level Synthesis Conference, IEEE*, 2012.

[104] G. C. Necula. Proof-carrying code. *In 24th symposium Principles of Programming Languages*, pages 106–119, 1997.

[105] G.C. Necula. Translation validation for an optimizing compiler. *In Proceeding PLDI'00 Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 83–94, 2000.

[106] V.C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard. Formal verification of compiler transformations on polychronous equations. *In Proceedings of 9th International Conference on Integrated Formal Methods IFM 2012, Springer Lecture Notes in Computer Science*, 2012.

[107] V.C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard. Formal verification of automatically generated c-code from polychronous data-flow equations. *Accepted at IEEE International High-Level Design, Validation and Test Workshop HLDVT 2012, California, USA*, 2012.

[108] V.C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard. Formal verification of synchronous data-flow program transformations toward certified compilers. *In Frontiers of Computer Science, Special Issue on Synchronous Programming, Springer*, 2013.

[109] V.C. Ngo, J-P. Talpin, and T. Gautier. Efficient deadlock detection for polychronous data-flow specifications. *In Proceedings of the 2014 Electronic System Level Synthesis Conference, ESLsyn 2014, San Francisco, CA, USA*, 2014.

[110] N.Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub, 1993.

[111] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg, 2005.

[112] Polychrony on Polarsys. An eclipse project of the polarsys industry working group. http://www.polarsys.org/projects/polarsys.pop, 2014.

[113] K.J. Ottenstein. An intermediate program form based on a cyclic data-dependence graph. *CS-TR 81-1, Dept. of Computer Science, Michigan Technological Univ., Houghton, MI*, 1982.

[114] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *In Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Programming Development Environments*, 9(3):177–184, 1984.

[115] S. Pinchinat, H. Marchand, and M. Le Borgne. Symbolic abstractions of automata and their application to the supervisory control problem. *In INRIA Technical Reports No 1279*, pages 1–29, 1999.

[116] G.D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

[117] A. Pnueli and A. Zaks. Validation of interprocedural optimization. *In Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2008), Elsevier*, 2008.

[118] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *In B. Steffen, editor, 4th Intl. Conf. TACAS'98*, pages LNCS 1384:151–166, 1998.

[119] A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation: From signal to c. *In Correct Sytem Design Recent Insights and Advances*, pages LNCS 1710:231–255, 2000.

[120] W. Polak. Compiler verification and specification. *Lecture Notes In Computer Science*, 124, 1981.

[121] GNU Project. Gcc bugzilla. http://gcc.gnu.org/bugzilla/, 2014.

[122] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. *In International Symposium on Programming, LNCS 137, Springer Verlag*, 1982.

[123] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. *In Proc. 5th Int. Symp. in Programming*, 1981.

[124] W. Reisig. Petri nets. *New York: Springer Verlag*, 1985.

[125] M.C. Rinard. Credible compilation. *Technical Report of MIT Lab for Computer Science 776*, 1999.

[126] J. Rushby. New challenges in certification for aircraft software. *In Proc. 9th ACM Int'l Conf. Embedded Software, ACM*, 2011.

[127] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. *Oxford Programming Research Group Technical Monograph. PRG-6*, 1971.

[128] F. Sheridan. Practical testing of a c99 compiler using output comparison. *Software: Practice and Experience*, pages 37(14):1475–1488, 2007.

[129] Simulink. Simulation and model-based design. http://www.mathworks.com/products/simulink/index.html?s_tid=gn_loc_drop, 2014.

[130] STL. Standard c++ library reference. http://www.cplusplus.com/reference/, 2014.

[131] A. Stump and M. Deters. Smt-comp. http://www.smtcomp.org/2009, 2009.

[132] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equility saturation: A new approach to optimization. *In 36th Principles of Programming Languages*, pages 264–276, 2009.

[133] J.W. Thatcher, E.G. Wagner, and J.B. Wright. More on advice on structuring compilers and proving them correct. *Lecture Notes In Computer Science*, 94, 1980.

[134] TopCase. The open-source toolkit for critical systems. http://www.topcased.org/, 2013.

[135] J-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. *In ACM SIGPLAN Conference on Programming and Language Design Implementation*, 2011.

[136] D.A. Turner. A new implementation technique for applicative languages. *In Software: Practice and Experience*, 9:31–49, 1979.

[137] D. Weise, R.F. Crew, M.D. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. *In 21th Principles of Programming Languages*, pages 297–310, 1994.

[138] Z. Yang, J-P. Bodeveix, M. Filali, K. Hu, and D. Ma. A verified transformation: from polychronous programs to a variant of clocked guarded actions. *In 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES'14)*, 2014.

[139] L. Zuck, A. Pnueli, and R. Leviathan. Validation of optimizing compilers. *Technical Report MCS01-12, Weizmann Institute of Science*, 2001.

[140] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A translation validator for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 65(2), 2002.