SILC: SPRITE Input Language with C(++)

Master's Thesis

by

Philip Hölzenspies

Committee:
dr. ir. Gerard Smit
ir. Michèl Rosien
dr. ir. Jan Kuper
prof. dr. ir. Thijs Krol

Cover art by Erik Hagreis

University of Twente, Enschede, The Netherlands
April 29, 2005

Preface

Programming paradigms and compiler techniques have been one of my specific interests for quite a while now, so when Gerard Smit offered me this assignment, it immediately felt like it was going to be a lot of fun. “Back then” I was a big fan of c++ and was very enthusiastic about the opportunity to study it closer and try and design new models of compilation for it. I say “back then,” because these past few months of research have brought surprise after surprise with respect to the behavior of the language. Not all of them nice.

However, the process of researching itself *was* very enjoyable. Very much so, because of the feisty arguments with Michèl Rosien—in which I was always much too formal—and sparring at the whiteboard with Jan Kuper—where I was never formal enough. I thank them both for these many hours very well spent. Gerard Smit offered me this assignment and for that alone I am grateful, but more so, he stayed calm when I stressed out about deadlines and such. Thijs Krol gave me a real eye-opener halfway into my research, when he completely scattered and rearranged my semantic understanding of SIL, which eventually helped me a lot. After all, SIL is his work and the thought that I completely understood all levels of abstraction after reading a concise summarizing paper left me sadly mistaken. Working with the people in my committee was all in all a highly enjoyable and educational experience.

Outside of the committee there are too many people I would like to thank to do that here, so I will simply say I am grateful for all the love I receive daily from friends and family. However, two people have helped in such exceptional ways that they *do* need to be mentioned explicitly. Erik Hagreis has put a lot of the time he did not have into the design of the cover of this thesis. The result is something to be seen. Likewise, Mark Westmijze has sacrificed a lot of his time to create a presentable demo for me. I must say, his demo has turned out to be quite stunning.

I hope this thesis is useful and reading it will be enjoyable. Writing it certainly has been.

Philip Hölzenspies
April 2005

Contents

List of Figures

1 Introduction

The SPRITE Input Language (SIL) was designed to enable transformational design in VHDL implementations of electronics. It gives a graphical view of the control- and data flow through the entire design, because it is itself a Control- and Data Flow Graph (CDFG), or actually a hypergraph. To allow a designer to get a clear overview, it allows for hierarchical abstractions that hide complexity, but still guarantee correctness.

The assignment for this thesis was to extend SIL with types and operations that would allow the mapping of a greater subset of c(++) to it. This need arose from the observation that designs are often implemented first as a runnable specification. For this purpose c++ is used quite often. Moreover, applications in software sometimes need reimplementation in hardware, because they fail to meet (increased) performance criteria. Both of these cases call for an automated translation, because the manual methods often used are error prone and time consuming.

The '++' is parenthesized, because objects and other equally complex extensions c++ introduces over c are not (yet) included in the supported subset. However, mixture of statements and declarations is, which will be shown to have quite significant implications for the modeling extensions required. Moreover, would there turn out to be conflicts between c [8] and c++ [5] then the c++ standard was chosen to be the modeling goal.

There have already been some attempts at extending SIL, but they have not yet been completely successful. Considering that these attempts provided the most important basis for this thesis. A concise description of these extension attempts was drawn up as an internal report by Thijs Krol and Bert-Steffen Visser. Since this report is not available outside of the Embedded Systems group of the University of Twente and the work presented here relies so heavily on it, it has been included as an appendix (see appendix A).

Because of the limited time available for this assignment, a subset of the uncovered language features was chosen. Therefor, the extensions presented here are restricted to pointers and jumps. These two language features were deemed 'most missed' and also occur in many (if not most) other imperative languages.

The structure of this thesis is as follows: As it turned out the terms used in the

context of SIL varied from one conversation to the next, chapter 2 describes the total process of translation and transformation and provides definitions of the terms used in the following chapters. Next, the key findings from experimentation and review of the standard with respect to pointers (chapter 3) and jumps (chapter 4) are presented and resulting design choices explained. Chapter 5 gives the complete formal definition of the extension to the model. Because of limitations found with respect to transformability of SILC, chapter 6 describes an implementable optimization using annotation. Finally, chapter 7 gives a roundup of the results and recommendations for future work.

2
The Process

This chapter deals with the process of translation. The conventional way of compiling c++ is described briefly in section 2.1. A *very* brief and informal introduction to SIL is given section 2.2 (a much more extensive explanation can be found in appendix A). Transformational design is introduced next (2.3) and the consequences for compilation are discussed. Section 2.4 shows how these consequences have been incorporated in the work process used when working with SILC. Finally, the process of transforming and the nature of transformations is laid out in section 2.5.

2.1 Compiling c++ directly

Commonly, programs are said to have definitions at compile- and run-time. The programmer/designer must be familiar with these concepts in order to deliver proper work. The vast majority of compilers work in some sort of continuous mode and allow for very little (if any) interaction. They are programs themselves that take program code as input, process it and deliver their output. This output can be machine executable code for whatever platform, byte-code for some sort of evaluator or some sort of file used by machines to produce hardware.

Of course, every such output requires its own compiler, because many of these different types of output require highly specialized processing (if any level of optimization is desired). There are many compiler suites (e.g. the GNU Compiler Collection) that group different front- and backends and unify the intermediate data structures and processors. However, all these compilers traditionally work in a batch oriented fashion. Any unrecoverable error (i.e. by the compiler itself) will result in a termination of the compilation process and incomplete output, or none at all.

Real errors should only occur on erroneous input, but failed optimization attempts just result in the original (unoptimized) code they were fed. These optimizations are commonly algorithm based (as opposed to artificially intelligence based) meaning that many are either NP-complete or will not recognize all optimizable structures.

All compilation parameters and hints a designer can give with respect to these opti-

mizations have to be chosen offline, i.e. before feeding the program to the compiler. It is always possible to change the hints and parameter values and to recompile a program when the original values result in an unsatisfactory compilation.

2.2 SIL

SIL stands for SPRITE Input Language and was designed by Thijs Krol et al as an intermediate format for High Level Synthesis. It needed to have the expressive power and interpretation ease to be applicable to Transformational Design (see section 2.3). SIL is based on the notion of hypergraphs. In a hypergraph, a hyperedge connects one set of nodes to another—as opposed to a single node to another—defining (in the case of SIL) a relation between all the nodes it connects. Actually, nodes are points in the hypergraph that receive a *valuation* and the hyperedges define relations between these valuations of nodes. For brevity, hyperedges and -graphs are often referred to as edges and graphs, since 'normal' graphs are never relevant to this text.

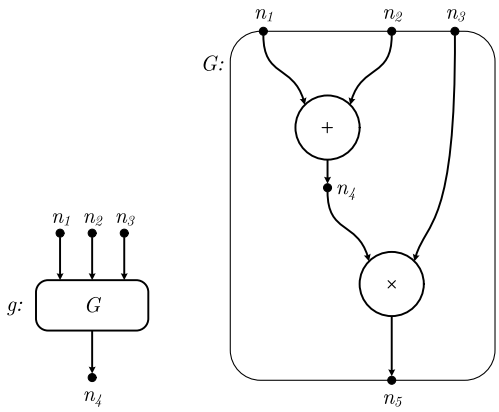
As a rather simple example, consider the graph depicted in figure 2.1a. Nodes, drawn as small circles[1], are connected by hyperedges. Hyperedges are drawn as boxes with arrows to and from nodes. The nodes with arrows to a hyperedge are referred to as the hyperedge's *input nodes* and the nodes pointed to by the arrows from a hyperedge are considered its *output nodes*. Furthermore, note that the hyperedges are labeled with *two* labels. The one on the outside (*g* in the example) is the name of this specific hyperedge and the one in the box (*G* in the example) is the name of the hyperedge's definition. A definition can be given in terms of either another graph, in which case the hyperedge is referred to as *complex*, or a primitive function, in which case the hyperedge is considered *primitive*. Primitive hyperedges can be considered to be defining themselves. As a convention, primitive hyperedges are often drawn as (large) circles or ellipses. Nodes are also labeled with their names, but note that both node and edge names are only a way to reference a specific node or edge, i.e. a name is not intrinsically part of the node or edge it refers to.

The hyperedge (often simply referred to as 'edge') in the example is complex and, as such, there must be a definition in terms of another graph for this edge. It is depicted in figure 2.1b. The graph's name is shown similar to how names of edges are depicted. It turns out *G* is a graph that has two primitive edges ('+' and a '×'). Possible definitions for these edges are:

$$\begin{aligned} + &: \mathcal{V}(i_1) + \mathcal{V}(i_2) = \mathcal{V}(o_1) \\ \times &: \mathcal{V}(i_1) \cdot \mathcal{V}(i_2) = \mathcal{V}(o_1) \end{aligned}$$

Where i_n refers to the n^{th} input node and o_m to the m^{th} output node. The $\mathcal{V}$ function maps nodes to their values and as such these definitions (and the edges that reference them in the graph) define relations on valuations of nodes. Note that these operations happen to be commutative and thus explicit naming of the in- and output nodes is not very relevant. When it is, a graph is usually shown with *only* the primitive edge and its in- and output nodes, naming the nodes. When it is unclear to what input a node is connected, the arrow connecting it may be labeled with the name of the input. The same holds for outputs. Note that the scope of these names is limited to the edge that

[1] In SIL only filled circles are used for nodes, but SILC will introduce a state space as 'special' data, so to differentiate between the two, nodes representing state will be drawn unfilled.



(a) Graph with a single complex edge

(b) Defining graph for G

Figure 2.1: SIL example graphs

requires them, i.e. the fact that the output of *graph G* is named n_5 does not conflict with the fact that the *edge g* gives its output to node n_4[2].

A more formal and thorough description of SIL is available in appendix A.

2.3 Transformational Design

The idea that all designer supplied compiler directives are given before actual compilation takes place means the designer must be very aware of the capabilities of the compiler and must be familiar with the compilation process. It is possible the designer misses a lot of characteristics of a given program, simply because sometimes relations seem highly complex, when they are not. Moreover, parts of a program might seem very expensive (in terms of execution time) prior to optimization, but might be reduced to something very cheap. This is usually discovered during profiling, but it seems desirable to have such information before producing the final output to avoid the need for recompilations.

Moreover, in the context of hardware/software co-design the strict division between compile-time and run-time forces early choices with respect to the separation of hard- and software. By introducing a new and interactive phase in the process of compilation, *transform-time*, many of these choices can be made at a later stage when more information is available. SIL was developed as an intermediate representation—initially for high level synthesis [4,9,10]—for operation during transform-time. It gives a graphical representation of the program being compiled and should thus provide a more intuitive perspective on the functionality of the program to the designer. Another strength of SIL is that it can be mapped to a wide variety of different target languages or other outputs. This means that e.g. the choice which parts of a program to implement in hardware can be postponed to a point where it is known what parts *really* are computationally intensive.

[2] Actually, all names are prefixed by their instantiating environment, so the n_5 of the definition is called $g.n_5$ in the more global scope shown in figure 2.1a.

This transform-time—which is essential in transformational design—can be seen as explicit and interactive compilation. A source language is compiled to the representation used (SIL). Next, transformations that are proven to leave the external behavior unchanged are performed as per the instructions of the designer [4]. After the designer performs all the transformations he or she deems required, the complete SIL-graph can be output to a certain output format, or parts can be selected one by one and be outputted much the same way.

SIL can be translated to a multitude of output languages. As said before, it was primarily intended for high level synthesis and was thus designed to translate from and to languages such as VHDL. It can be argued that SIL is itself a functional programming language and thus *translations to and from other functional languages* are very intuitive.

A relatively new application for transformational design involves translating runnable specifications directly to implementations. It is common practice to first deliver such a runnable specification to a client to test the interpretation of the design requirements and specifications. Currently, this is often done manually, which is time-consuming and error-prone, so automatic ways of translation are desirable. The basic use case of this application involves translating c++ programs to, e.g. VHDL. A close inspection of SIL shows that it has insufficient notion of 'state' to really be able to translate imperative languages to it. With the extensions described in this thesis (SILC) it is now possible to map from and to imperative languages.

Considering the above, it seems worth exploring whether it is possible to adapt SIL in such a way that it *will* facilitate translation to and from other languages. In the most optimistic view, it may even be possible to use SIL as a *universal translator*. The extensions to SIL in SILC will be restricted to the imperative languages.

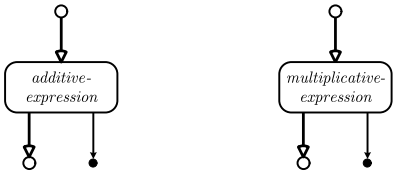
2.4 Compiling C++ through SILC

SILC stands for 'SPRITE Input Language (SIL) with C(++)-extensions' and it adds an extended notion of *state* to SIL. This notion of state will be treated extensively in the following chapters, but a brief introduction is useful here. The state is modeled simply as a very specific type of data, with its own primitives. These primitives basically read and write data[3] from and to addresses in the *statespace*. The statespace is the collection of all possible states, but the term statespace is also used loosely to indicate all states occurring in the program. In c++ every statement potentially (and presumably) alters the state implying that (at least initially) the flow through the program is indicated by the changes in the statespace.

For the 'correctness by construction' criterion[4] [4, 10] to be applicable, the initial input of the transform-time interaction must be correct. This means the translation from c++ to SILC must be guaranteed to be correct. To obtain a verifiable translation, the rules of translation should relate as closely as possible to the c++ grammar [8]. By using template instantiation with a separate template for every grammar rule, the correctness becomes testable on a per-rule basis.

[3] Since evaluation does not have to be completed for data to be data, they actually read and write subgraphs.

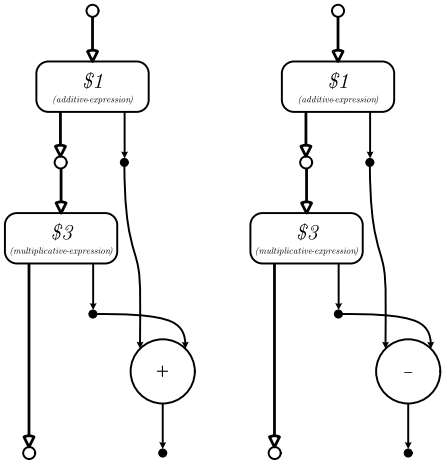
[4] Correctness by construction states that given correct input, any sequence of proven transformations (i.e. transformations that are proven to leave the external behavior unchanged) will render correct output, i.e. having the same behavior.



(a) Template for the *additive-expression* grammar rule

(b) Template for the *multiplicative-expression* grammar rule

Figure 2.2: Examples of common templates



(a) Addition template

(b) Subtraction template

Figure 2.3: Template for the second and third clause of the *additive-expression* grammar rule

As an example of this mechanism consider the following segment of the c++ grammar [5, 5.7/1]:

additive-expression:
  multiplicative-expression
  additive-expression + *multiplicative-expression*
  additive-expression - *multiplicative-expression*

The complete template for the grammar rule for an additive expression is depicted in figure 2.2a. In the simplest case, an additive expression is just a multiplicative expression. If this is the case, the inputs of the additive expression template can just be connected to the inputs of the multiplicative expression template (fig. 2.2b). The $\$n$ notation is taken from bison's [2] denotation of the semantic value of the n^{th} symbol of the relevant clause; the first rule only has one symbol, so $1 denotes the result of parsing the multiplicative expression.

If an additive operator (+ or -) is found, either the second or the third rule should be chosen depending on the operator. A template can be formulated for both of these rules (fig. 2.3). There is now a one-to-one correspondence between the set of templates given here and the *additive-expression* grammar rule.

Having said that SIL graphs can simply be seen as functional programs (2.3) and that transformations preserve external behavior[5], transformations are in essence offline evaluation. The problem is, of course, that the complete evaluation can rarely occur at transform-time[6], so not all run-time information is available yet. Leaving the unknown as is, it still often is possible to transform the known parts of the program. Transformations should thus abstract away from the unknown.

An observation is required with respect to what is and what is not unknown. Any program is in itself incomplete if part of its *functionality* is unknown, so what *is* allowed to be unknown is restricted to (input)variables. These are modeled either as inputs into the graph, or as unknown constant edges[7] [10]. In the first case, no special implications arise, but in the second case it is possible to abstract away from this oblivion by quantifying over all possible (and contextually legal) constant edges.

The following concepts are hereby introduced:

- Let N be a set of nodes. An *ordered hyperedge* e on N consists of a (possibly empty) set ι of *input nodes*, and a (non-empty) set o of *output nodes*, i.e.:

  $$e \quad = \quad \langle \iota, o \rangle.$$

  Both ι and o are supposed to be ordered. For brevity, the term *edge* will be used instead of ordered hyperedge. E will commonly be used to denote a set of edges.

  If the set ι is empty, e is called a *constant edge*.

- A *template* t consists of a set of nodes N and a set of hyper edges E on N.

  $$t \quad = \quad \langle N, E \rangle$$

  An alternative term for template is *hyper graph*.

  A set of templates is indicated by T.

- Let L be a set of labels. A *labeling function* f is a function of type $E \rightarrow L$, i.e. f assigns labels to hyper edges. The function f may be partial, i.e. not all edges in a template need to be labeled.

  Labeled, non-constant edges are often referred to as (SILC-)*operators*.

- A *labeled graph* C is a 3-tuple

  $$C \quad = \quad \langle N, E, f \rangle$$

  where $\langle N, E \rangle$ is a template, and f a (possibly partial) labeling function of E.

  The template $\langle N, E \rangle$ will be called the *underlying template* of the labeled graph C.

- Let $t = \langle N, E \rangle$ be a template, and f, f' two labeling functions of E. Let $\varepsilon \subseteq E$ be a set of edges of t such that both f and f' are total on ε.

  Two labeled graphs $\langle N, E, f \rangle$ and $\langle N, E, f' \rangle$ are *ε-equivalent* (or *ε-isomorphic*) if f and f' assign the same labels to all edges in ε. More formally:

  $$\langle N, E, f \rangle \cong_\varepsilon \langle N, E, f' \rangle \quad \Leftrightarrow \quad f \upharpoonright \varepsilon = f' \upharpoonright \varepsilon$$

  where $\upharpoonright$ restricts the functions f, f' to the set ε.

[5] It is still possible for a designer to make changes to a program in this phase and thus to apply behavior altering transformations, but this is an *explicit design choice* and should not occur automatically

[6] As a matter of fact, only if the entire program is constant can it be fully evaluated offline.

[7] Constant edges have zero inputs and all outputs are given constant values.

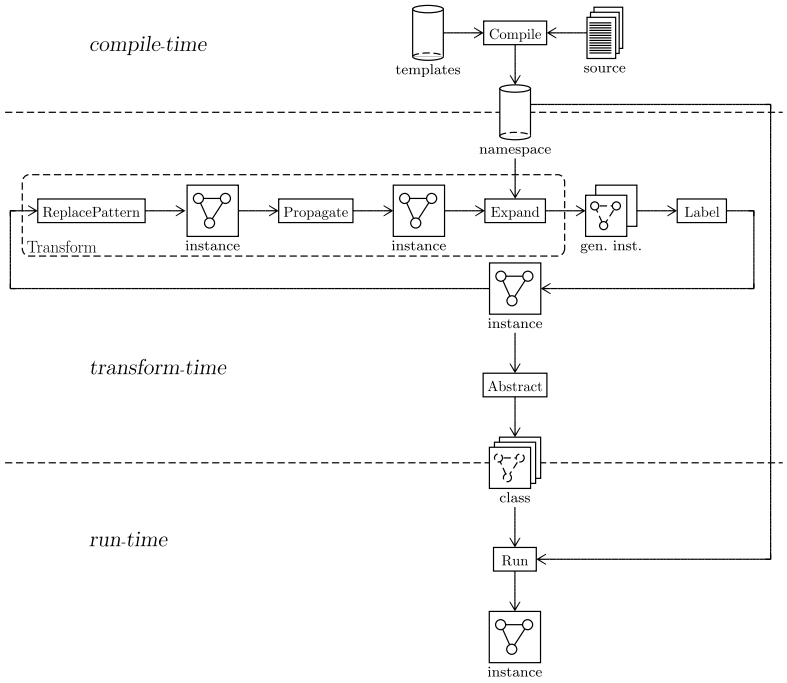


Figure 2.4: Transformational design process

- A *graph class* is the set of all labeled graphs with the same underlying template, that are ε-isomorphic for a given set ε. A graph class will be denoted by $\langle N, E, f, \varepsilon \rangle$, where f is a representative of the set of all ε-isomorphic labeling functions on E.

  Often, we will simply speak of *class* instead of graph class.

- A labeled graph G in a graph class $\langle N, E, f, \varepsilon \rangle$ is an *instance* of that class, if the labeling function f_G of G is total on E and injective on $\delta = E \setminus \varepsilon$. Remember that $f_G \restriction \varepsilon = f \restriction \varepsilon$.

  The graph instance G will be denoted as $\langle N, E, f, \delta \rangle$. Note, that the difference in notation between a graph class and a graph instance only consists of the symbols ε and δ. In the following text these different symbols (and accented variants like δ') will be used to indicate whether a class or an instance is being discussed.

In practical terms, the quantification over (constant) edges mentioned above comes down to quantification over all labelings the edge can have. The vast majority of labels is known after compilation, because many elements in a program are in one way or another constant (operators, functions, hard coded values, etc). In terms of the definitions given above: a class' set of edges with fixed labelings ε will include the majority of the class' complete set of edges E.

Templates are instantiated during compilation of the source language. The result is a set of (named) graph classes, i.e. the namespace. The labelings of all edges *not* in ε require run-time input, so they will not be available at transform-time. By choosing a *unique* random value for these unknown labels, it is possible to calculate relationships at transform-time without having to predict their actual run-time values. The chosen

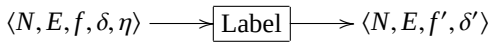


Figure 2.5: Arbitrary labeling function block

values are simply placeholders and do not have any semantic value for the final run-time program, but at least they are guaranteed to be chosen from the graph class. Basically, during transformation *any* labeling function f' may be used as long as it preserves the ε-isomorphism with the *representative* labeling function in the namespace *and* a track record is kept to indicate what edges have been labeled randomly (δ).

These random labels are retracted by abstraction to deliver a graph class that can be instantiated at run-time to get the 'real' labelings. *Running* a program is now reduced to *choosing* a specific instance of the graph class. Hence when run, a graph is chosen from the class with labeling function g and of course inputs—if required—are given by means of a valuation function $\mathcal{V}$ of the input nodes. The entire process—from source to execution—is depicted in figure 2.4.

2.5 Transformations

In the diagram in figure 2.4 the complete intelligence of transformation is split up into three categories: structural, operational and hierarchical. These transformation categories and the transformations required[8] for the guarantee of consistency will be explained briefly in this section to gain some sort of intuition of the transformation process.

2.5.1 Labeling

The labeling function block (fig. 2.5) appoints trivial, but unique labels to unlabeled edges. It takes as an argument what is loosely called a 'generic graph instance', because it is less defined than a graph instance, but more defined than a graph class. It is actually a graph instance with some edges that have not been dummy labeled yet. This means it has an extra set of edges $\eta \subseteq E$ that still require labeling. Thus all significantly labeled edges in E are $E \setminus (\delta \cup \eta)$.

Labeling does not change the structure of the graph itself, i.e. $\langle N, E\rangle$ is unchanged, but it chooses a new labeling function in such a way that all chosen and fixed labelings from the previous labeling remain and new labelings are chosen for the edges in η. Hence

$$\begin{aligned} f \restriction (E \setminus \eta) &= f' \restriction (E \setminus \eta) \\ \delta' &= \delta \cup \eta \end{aligned}$$

Note that this means that all edges in δ retain the labeling assigned to them by f under f'. The labeling should guarantee that $f' \restriction \delta'$ is absolutely injective.

2.5.2 Abstraction

The labels introduced at transform-time have been used to observe equality, but have no run-time significance. Before running a program, the assumptions made for offline

[8] Label and Abstract.

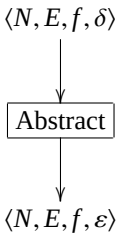
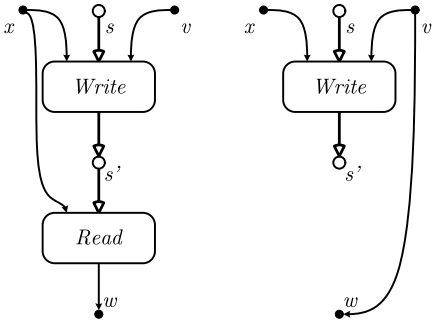


Figure 2.6: Abstraction function block



(a) Pattern to match (b) Result of transformation

Figure 2.7: Example of pattern-based transformations

evaluation should thus be retracted, i.e. the dummy labels assigned to those edges that were not significantly labeled in the graph class should be 'removed'. This happens by abstraction from the instance resulting from transformation to a class.

Note that abstraction—as depicted in fig. 2.6—may very well leave the labeling function intact, because in the definition of a graph class, f is a *representative* labeling function for the ε-equivalence. Retracting the dummy labeling comes down to reversing the indication from which edges are labeled randomly (δ) to which edges are labeled significantly (ε):

$$\varepsilon \quad = \quad E \setminus \delta$$

2.5.3 Structural transformations: pattern replacement

The category of structural transformations concerns mostly pattern replacements, i.e. the replacement of parts of a graph, based on the *structure* of those parts. Patterns can be defined to describe replaceable structures in a graph. Consider as an example the case that a value `v` is assigned to variable `x` in the state and directly after, the same variable is read (as `w`), than this may be described as a pattern (fig. 2.7a) that can replaced with the behavioral equivalent which still write in the state, but that just copies the value from `v` to `w` directly (fig. 2.7b).

This kind of replacement can be generalized for a sizable number of patterns. SILC actually has a few patterns predefined, but any and all pattern replacements should follow directly from the definition of the model's operators. This does not constitute operational transformation (2.5.4), because the input values themselves are not considered for the transformation.

Figure 2.8: Structural transformation function block

Figure 2.9: Operational transformation function block

Equality between input values *is* of course relevant (from the example above, the address inputs were compared to see they both received *x*), but in the graph context, equality follows from coming from the same input node. Two inputs can very well be equal when coming from different nodes, but if so, further transformations will—in most cases—unify these different nodes.

Looking at the structural transformation function block (fig. 2.8) a few assertions can be made: both nodes and edges may be destroyed or introduced, so there is no general constraint that can be given on the relations between N and N' and between E and E'. What *can* be stated is that any edges that remain in the output have unchanged labelings, i.e.

$$f \upharpoonright (E \cap E') \quad = \quad f' \upharpoonright (E \cap E')$$

which implies that

$$\delta' \quad \subseteq \quad \delta$$

2.5.4 Operational transformations: propagation

Referential transparency is the property of operators and functions in general that guarantees that constant arguments imply constant results. All operations in SILC are referentially transparent, since the complete state can be an argument of an operation. This observation makes available transformations that take into account knowledge of the definition of operators. When an instance contains operators with exclusively constant inputs, it can be replaced by a set of constant edges; one for each of its outputs.

The function block for this type of transformations is shown in figure 2.9. The same assertion as made for structural transformations holds with respect to the labeling functions, so

$$f \upharpoonright (E \cap E') \quad = \quad f' \upharpoonright (E \cap E')$$

Another important assertion that holds for this category (since it only involves replacing operations with constants for their outputs) is that there will be no introduction of new nodes, hence

$$N' \quad \subseteq \quad N$$

2.5.5 Hierarchical transformations: expansion

Actually the hierarchical transformations are twofold: expansion *and hiding*, but hiding is not very relevant in automated transformations: When the designer is trying to get a cleaner picture of the state the—partially transformed—program is in, hiding can

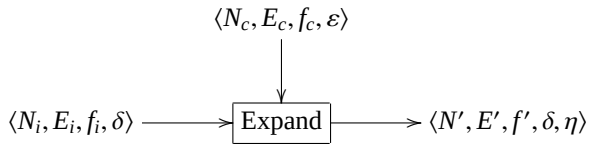


Figure 2.10: Hierarchical transformation function block

be a very useful tool, but the automated transformer will only look at relatively small localities and has little to gain from hiding. The semantics of both types of hierarchical transformations are described very clearly in [10, section 2.2], especially the renaming of nodes and edges to unique new names with the exception of in- and output nodes. Only expansion will be treated, but hiding should follow intuitively from the process description given here and the semantics.

Without a complete definition, a homomorphism [1, section 1.4.1] ϕ is used to rename nodes ($\phi_0 : N \to N'$) and edges ($\phi_1 : E \to E'$) conforming to the semantical definition of expansion[9]. With this homomorphism, consider the function block depicted in figure 2.10. The arguments of the expansion function are the graph instance in which an expansion is necessary and the graph class that defines the edge that is to be expanded. The function results in a generic graph instance (as described in 2.5.1).

Given the definition of a *relational image*

$$f(\!|S|\!) \quad = \quad \{f(s) \mid s \in S\}$$

the following assertions hold (where e is the edge being expanded):

$$\begin{aligned} N' &= N_i \cup \phi_0(\!|N_c|\!) \\ E' &= (E_i \setminus e) \cup \phi_1(\!|E_c|\!) \\ f' &= f_i \cup (\lambda\langle d, r\rangle.\langle \phi_1(d), r\rangle)(\!|f_c|\!) \\ \eta &= \phi_1(\!|E_c \setminus \varepsilon|\!) \end{aligned}$$

In other words:

- The resulting set of nodes is the set of nodes from the instance expanded with the appropriately renamed nodes from the class being expanded (instantiated).
- The resulting set of edges is the set of edges from the instance expanded with the appropriately renamed edges from the class being expanded.
- The resulting labeling function is the labeling function from the instance expanded with the labeling function from the class, where the latter's domain is renamed according to the renaming of nodes.
- The set of edges that require (dummy) labeling after this transformation is the set of appropriately renamed edges from the class *not* significantly labeled.

[9] Hence, φ_1 renames all edges in the graph being expanded such that all their names are unique. Nodes are renamed by φ_0 in such a way that all input and output nodes are given the names of the nodes to which they are connected. 'Internal' nodes are given unique names.

Summary

Transformational design is based on ‘correctness by construction’, which can only be accomplished by provable translations from the source language to the first instance that will be transformed. To accomplish this, the translation from c++ to SILC is based on template instantiation with a one-to-one correspondence of templates and grammar rules.

A new phase—transform-time—in the process is necessary to accommodate transformational design in which the designer interacts with a transformation tool to decide what transformations are to be performed.

The program representation of SILC—a hypergraph—is in fact a functional program. Transformations can therefor be considered offline evaluation of said functional program.

Since some information might not be available prior to run-time, dummy values are inserted that are known to be unique so that relations between nodes can be observed without having the actual values themselves.

3 Pointers

In this chapter, some problems arising from the use of pointers are described and solutions to deal with these problems are proposed (3.1 through 3.3). These solutions will be shown to introduce new problems themselves and thus the chapter progressively describes (by iterative solving and examining the solution) the way to the final solution presented in chapter 5. The final section (3.4) a sizable example is given to illustrate the findings of this chapter.

3.1 Problem: Equality and loss of origin

Modeling pointers implicitly by their symbolic name, like any other variable, hides the context of the pointer in the implicit context. When pointers are offset from their base position, the expression itself is required (in its entirety) to determine the referenced location in memory. This means that widespread interdependence requires propagation of a potentially large subgraph through the graph to bring these interdependencies closer together, which becomes very hard when trying to transform it over a possibly infinite recursion.

Consider the example graph shown in figure 3.1 in which it is already determined that the state space does not change in X. For a very complex graph X, the relation between pointer b and address d can not be seen on a local scale, thus, to determine interdependence between pointers a and b, the transformation tool needs to track the full evaluation path of both pointers and see if these paths intertwine somewhere. There should be some way to propagate the *constant of the dependence*, no matter what the offset evaluates to or depends on, i.e. to partially propagate the eventual value of b.

Even a transformation that should be relatively simple becomes rather complex when using symbolic names as models for pointers. Propagating constants that are added to a pointer along the way turns tricky when the constants are not grouped, but added directly to the pointer. Figure 3.2a shows two constants being added to a pointer a. In order to perform constant propagation, the transformation of exchanging 3 and a is required first so as to obtain an edge exclusively connected to constant inputs, to

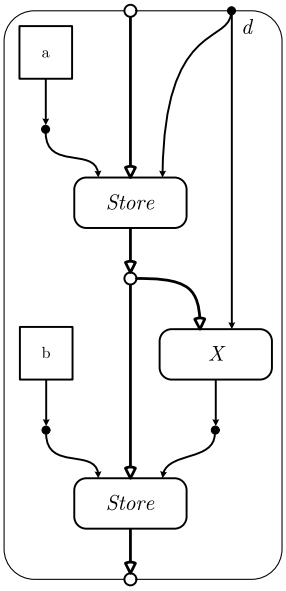


Figure 3.1: Dependence between pointers *a* and *b*, hard to decide

propagate said edge to a constant (fig. 3.2b).

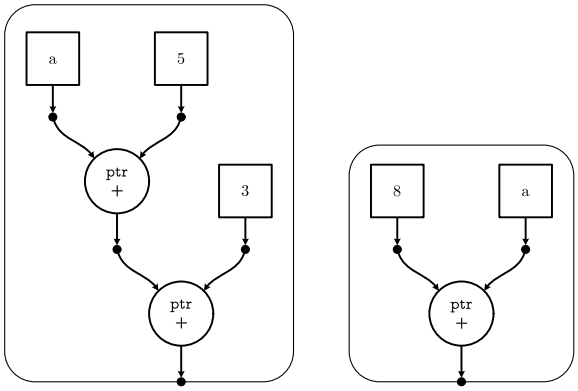
Another problem arising from symbolic representation of pointers is the *loss of origin*. When memory is allocated dynamically, it is not bound to a constant name, but rather its location is assigned to a pointer. Code snippet 3.1 illustrates the problem. The declaration of *a* also leads to its allocation and provides a fixed name for the memory reserved at that instance. Pointer *j* is assigned the address of an otherwise unnamed piece of memory. When *i* is incremented, the memory it points to still has its point of origin modeled by *a*, but when *j* is incremented, there is no means to point to the beginning of the original array.

Code snippet 3.1 Loss of origin

```
int  a[4],
    *i = a,
    *j = new int[4];
...X...
i++;
...Y...
j++;
```

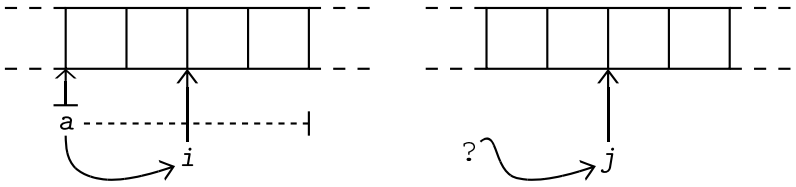
The problem here is that it is now impossible to use the model for bounds and leak checking[1]. It is probably possible to check the bounds of *i*, as it is formulated in relation to *a*, which is fixed (fig. 3.3a). However, to resolve how many times *j* could be decremented after its incrementation (fig. 3.3b), the entire evaluation hidden away in *X* and *Y* must be evaluated, which can—needless to say—become very complex.

[1] This might not necessarily be required of the model, but if the ability to perform these checks is available at little or no extra cost, it is worth examining.



(a) Unfortunate ordering (b) Optimized result

Figure 3.2: Constant propagation in pointer arithmetic



(a) Bound pointer (b) Afloat in “mid-memory”

Figure 3.3: Memory in terms of the pointers to it

3.1.1 Solution

Pointers require a representation within the model to allow for constant propagation, but it is important to note that this propagation only occurs through a limited arithmetic. Basically, pointers can be assigned, added to or subtracted from. Furthermore, observe that *any variable name is in effect a pointer to the actual variable*, albeit that it is dereferenced at compile-time.

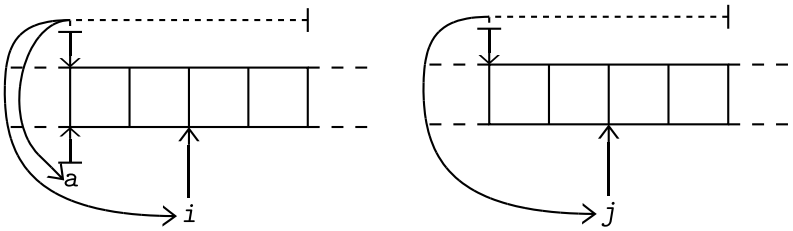
Concretely, two requirements have to be met:

- Origin reference: Any address must carry in its representation a reference to the beginning of the block of memory it points into. The representation should not limit the model to specific architectures, so the relation to physical addresses should be abstracted away from.

- Constant propagation: Pointer arithmetic should be propagable as much as possible, meaning that at least every operation with constant arguments should be propagable to a new constant.

These requirements are met by modeling pointers as tuples of a reference to the allocation of the memory pointed into and an offset in terms of the smallest addressable entity[2]. This gives memory allocations an autonomous identity that does not vary with transactions on named variables. In the example of code snippet 3.1, using this new

[2]On most stack machines, this would be byte-level, but when transforming to synthesis, this could very well be bit-level.



(a) *a* remains fixed, but no longer bounds *i* (b) No longer unbound

Figure 3.4: Memory in terms of autonomous identities

model simply reformulates the boundaries of *a* and *i* and gives a proper formulation of *j* (fig. 3.4).

By changing the modeling of a pointer, the model for the state space changes as well, because the symbolic names are no longer connected to locations in memory. Hence, there must be some sort of symbol table, which has locality and should thus be modeled in the state space. Informally, this leads to the following definitions:

$$\begin{aligned} \texttt{address} &= \texttt{(allocation\_index, offset)} \\ \texttt{state space} &= (\underbrace{\texttt{\{(symbol, address)\}}}_{\text{symbol table}}, \underbrace{\texttt{\{(address, data)\}}}_{\text{heap}}) \end{aligned}$$

3.1.2 Observations

These new "systematic addresses" are globally unique and can not be overwritten. As a direct consequence they are scope independent (as is the case with 'real life' memory). The symbol table as specified above would only be capable of modeling scoping if it was treated as an *ordered* set, where the first occurrence of a symbol is the symbol in the 'current' scope [3].

3.1.3 Consequences

With the new definition for the state space, pointer arithmetic allows for constant propagation explicitly by observing that there can not be any arithmetic function that projects one pointer onto another if they are not related to the same allocation (i.e. point into the same block of memory). Formally:

$$\begin{aligned} &\forall (x,p),(y,q) : \texttt{address} \mid x \neq y \bullet \\ &\quad \nexists f : (\texttt{address} \longrightarrow \texttt{address}) \bullet f(x,p) = (y,q) \end{aligned}$$

The expression depicted in fig. 3.2a can now be easily propagated, because

$$\begin{aligned} \texttt{a + 5 + 3} &= (\texttt{a}_{\texttt{alloc}}, \texttt{a}_{\texttt{offset}}) \texttt{ + 5 + 3} \\ &= (\texttt{a}_{\texttt{alloc}}, \texttt{a}_{\texttt{offset}} \texttt{ + 5) + 3} \\ &= (\texttt{a}_{\texttt{alloc}}, \texttt{a}_{\texttt{offset}} \texttt{ + 8)} \end{aligned}$$

Subtraction can intuitively be defined similarly (with the restraint that the offset remains positive). Subtractions of two pointers should subtract their offsets, but only when both pointers point into the same block of memory (otherwise the semantic value of the expression is void).

$$\begin{aligned} \texttt{a - 5} &= (\texttt{a}_{\texttt{alloc}}, \texttt{a}_{\texttt{offset}} \texttt{ - 5) iff a}_{\texttt{offset}} \texttt{ >= 5} \\ \texttt{a - b} &= \texttt{a}_{\texttt{offset}} \texttt{ - b}_{\texttt{offset}} \texttt{ iff a}_{\texttt{alloc}} \texttt{ == b}_{\texttt{alloc}} \texttt{ \&\& a}_{\texttt{offset}} \texttt{ >= b}_{\texttt{offset}} \end{aligned}$$

Multiplication and division on pointers are *not* defined in c++, so—observing arithmetic is limited to addition and subtraction—the assumption that arithmetic can be performed directly on the offset is valid. Consider snippet 3.2 as an illustration of invalid arithmetic on c++-pointers, because of possible rounding errors and overflow in multiplication and division.

Code snippet 3.2 Undefined behavior in pointer arithmetic

```
int a, *p = &a;
a  = (int) p;
a *= 2;
a /= 2;
p  = (int *) a; //is a==&a? probably not!
a  = *p;
```

3.1.4 Calibrated pointer arithmetic

In order to truly model c++'s real + operation, some form of typing is required. Observe the code equivalence depicted in snippet 3.3. For 32-bit architectures the `sizeof`-function applied to an integer (or the keyword `int`) would return 4, so adding this to a pointer shifts it for the width of one integer value. The incrementation of the `int`-pointer implicitly calls[3] a `sizeof(int)` and actually adds the result of this implicit call to the address represented by the pointer, instead of just incrementing the address in an untyped manner.

Code snippet 3.3 Implicit sizing in pointer arithmetic

```
int a[10],          ⎫     ⎧  int   a[10];
   *p;              ⎪     ⎪  byte *p;
p = a;              ⎬  ≡  ⎨  p  = a;
p++;                ⎪     ⎪  p += sizeof(int);
                    ⎭     ⎩
```

Typing should be used to align the data, i.e. to 'calibrate' pointer arithmetic. However, now that there are types, it becomes unclear whether the offset should be formulated in terms of the type of the pointer (I), or in terms of the smallest addressable entity (II) of the architecture. In (I), the physical equivalent of `(alloc_id, offset)` would then still require alignment and thus comes down to

$$\texttt{physical((alloc\_id,offset))} = \texttt{valueof(alloc\_id)} + \texttt{offset} \cdot \texttt{sizeof(typeof(alloc\_id))}$$

and the addition and subtraction operators are defined as follows:

$$\begin{aligned} \texttt{a + x} &= (\texttt{a}_{\texttt{alloc}}, \texttt{a}_{\texttt{offset}} + \texttt{x}) \\ \texttt{a - x} &= (\texttt{a}_{\texttt{alloc}}, \texttt{a}_{\texttt{offset}} - \texttt{x}) \ \texttt{iff}\ \texttt{a}_{\texttt{offset}} == \texttt{x} \\ \texttt{a - b} &= \texttt{a}_{\texttt{offset}} - \texttt{b}_{\texttt{offset}}\ \texttt{iff}\ \texttt{a}_{\texttt{alloc}} == \texttt{b}_{\texttt{alloc}}\ \&\&\ \texttt{a}_{\texttt{offset}} >= \texttt{b}_{\texttt{offset}} \end{aligned}$$

This method also requires an observation with respect to pointer casts. When casting from type `*a` to `*b` the offset of the pointer has to be recalibrated, i.e.

[3] Albeit that `sizeof` is a compile-time construct, so there is no call-overhead.

$$\texttt{cast(*a, *b, (alloc\_id, offset))} = \texttt{(alloc\_id,} \left\lfloor \frac{\texttt{sizeof(a)} \cdot \texttt{offset}}{\texttt{sizeof(b)}} \right\rfloor \texttt{)}$$

For (II), the calibration is in the operations, which are slightly more complex:

$$\begin{aligned}
\texttt{a + x} &= (a_{alloc}, a_{offset} + x \cdot \texttt{sizeof(typeof(a))}) \\
\texttt{a - x} &= (a_{alloc}, a_{offset} - x \cdot \texttt{sizeof(typeof(a))}) \texttt{ iff} \\
&\quad\ a_{offset} \texttt{ >= } x \cdot \texttt{sizeof(typeof(a))} \\
\texttt{a - b} &= (a_{offset} - b_{offset}) \cdot \texttt{sizeof(typeof(a))} \texttt{ iff } a_{alloc} == b_{alloc}
\end{aligned}$$

Consequently, the physical equivalent of `(alloc_id, offset)` and casting are now a lot more transparent.

```
    physical((alloc_id,offset))  =  valueof(alloc_id) + offset
cast(*a, *b, (alloc_id, offset))  =  (alloc_id, offset)
```

Code snippet 3.4 Differently typed pointers into the same block of memory

```
int   i = 0,
     *p = &i;
char *q = (char *) p;
...p...q...
```

The most significant difference between these two methods becomes apparent when considering the constraints these models impose on the model of the heap c.q. the operations on it. When an array is a chain of cells, where these cells have the width required to store a single array element, the (I) method is by far the most intuitive, but when pointers are cast, the arrangement of the memory they point to should change as well. This leads to problems when something like snippet 3.4 occurs. Here both `p` and `q` (and `i`, obviously) are used to address the same block of memory, but have different types. Alternatively, the operations on the heap could transform these type-dependant offsets to type-independent addresses and *then* perform their original function.

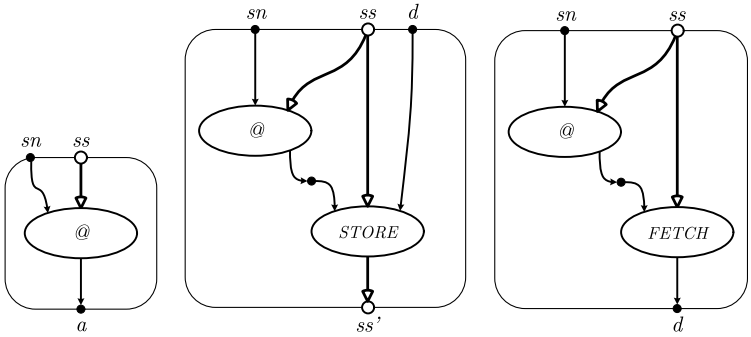
Code snippet 3.5 Out-of-phase pointers to the same block of memory

```
int   i[10],
     *p = i;
char *q = (char *) p;
q++;
p = (int *) q;
```

Besides the esthetic problems these solutions pose, (I) simply fails to model 'out-of-phase' pointers. Code snippet 3.5 illustrates the problem. After `q` is assigned the address stored in `p`, it is used to transpose `p` by the width of a `char`, but still pointing to segments of the width of `int`s. Note that `i` still points to the original location and `q` can shift to any part of the memory block. This situation can not be described by (I), hence (II) will be used to model pointers.

3.2 Problem: symbol table and scoping

As has been established in the last section, typing is a necessity. What remains uncertain, however, is how to store type information in the symbol table in the state space.



(a) Lookup (b) SymStore (c) SymFetch

Figure 3.5: Symbol table capable operations

The model given earlier works for a single scope, but becomes unpredictable when a nested scope overwrites a symbol.

3.2.1 Solution

Because symbol table and heap are now disconnected, though, it is possible to implement a scoping model in the symbol table and leave the heap as is.

```
state space  =  ({(symbol, [(type, address)])}, {(address, data)})
                                  scoping                heap
                symbol table
```

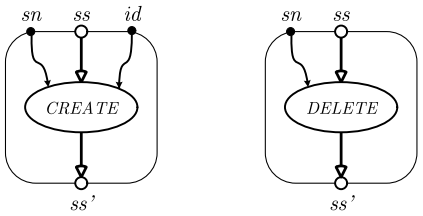
This is sufficient to model scope by observing that the topmost entry of the list of type/address-pairs models the definition in the 'current' scope [3]. Pointer casts from `*a` to `*b` are now transformations on the state space, albeit very simple transformations because the type-independent heap needs no transformations:

$$\begin{array}{rcl} \texttt{ss}_{in} & = & \texttt{(\{...,(x,[(a, ...),...]),...\},\{...\})} \\ & \Downarrow & \\ \texttt{ss}_{out} & = & \texttt{(\{...,(x,[(b, ...),...]),...\},\{...\})} \end{array}$$

3.2.2 Consequences

A very significant consequence (ignored earlier) that can no longer be avoided is the explicit need for primitive graph operations to perform state space transactions. Especially when dereferencing pointers, which requires an initial FETCH to obtain the address followed by a FETCH to obtain the data stored at said address. This implies that these FETCH's have different input types (symbolic name for the first and address for the second), which is not possible. Therefor, the definition of a lookup operation having an input for a symbolic name (*sn*) is required (fig. 3.5a).

In order to prevent an explosion of complexity of the graph (and of the transformations thereon), the definition of pseudo-primitive operations (*SymStore* and *SymFetch*, figs 3.5b and 3.5c) provides some containment. However, the CREATE and DELETE operations change (fig. 3.6) in their external typing (there is no sensible definition of a CREATE on an address as opposed to on a symbol), and thus correspond only to the *SymStore* and *SymFetch* operations in their external typing.



(a) Alternative CREATE (b) Alternative DELETE

Figure 3.6: Scoping operations adapted to comply with the symbol table model

As an illustration of just how much complexity increases, consider the translation of Codesniplet 3.6 into the graph depicted in figure 3.7.

Code snippet 3.6 Simple example of added complexity when using @

```
int  a = 0,
    *p = &a;
*p++;
```

3.2.3 Observations

Besides the added complexity of the graph itself, the transformations suffer from added complexity as well. The level of locality is decreased because of the distinction between the symbolic and the concrete representation of variables. In order to establish whether or not adjacent *Store* and DELETE operations are related their inputs can no longer be compared directly. The @-operation that results in the address given to the input of the *Store* needs to be in the locality being considered to be able to determine (in)dependence.

Furthermore, it is worth noting that the @-operation is a compile-time 'operation' and does not correspond to any run-time action or state change. Explicit @-operations never simplify or expand the capabilities of the transformations, because of their compile-time application (as opposed to the run-time relevance of transformation). Even more so, symbols themselves are inherently compile-time and have no meaningful representation at run-time (aside from debugging, of course). Therefor, if the graph is to model run-time behavior, it should *only include symbolic names as an assistance for recognition by the designer*, not as a semantic element.

Scoping was already implicitly modeled in the graph by the occurrence of the CREATE and DELETE operations. Having said that symbols do not exist at run-time, but only actual addresses are used, there is *no need to model scoping in the state space*. Besides the fact that transformations simply do not require this information to be explicitly available in the state space—because of their operation on edges in the graph—the final mapping of graphs to whatever target does not require any such explicit modeling either.

The only part of the state space that does actually have a representation in real life at run-time is the heap. Because the heap only uses actual addresses and not symbolic names, it is scope independent. This is in accordance with reality, where scoping does not exist at run-time either.

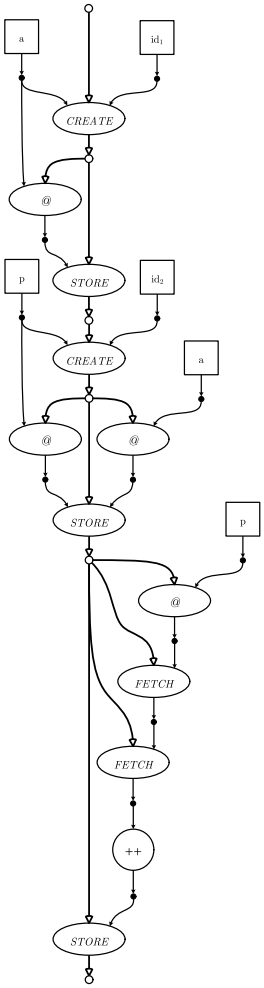


Figure 3.7: High complexity due to the added @-operation

Actually, if the model is to *cleanly* represent run-time behavior, the symbol table has no place at all in the state space.

3.3 Problem: No home for types

When the symbol table is removed from the state space, types can no longer be related to the symbols that where defined in terms of them. To allow for (castable) pointer arithmetic, however, the types are essential, because there has to be some way to know what specific operations have to be performed. Without types the offset problem mentioned earlier reoccurs. Hence, typing should be included elsewhere.

3.3.1 Solution

Types are, essentially, only required when alignment in the heap is relevant and thus they are only necessary when dealing with pointers, not when dealing with data itself, because data—in the graph—carries its type implicitly in its width. Pointers should thus carry a type and, since pointers are themselves data and thus carry their own type as any other data, the type carried should be that of what is pointed to.

This implies that any fully qualified address carries the type of what it addresses. This might seem counter-intuitive, but in fact, normally the runtime fetches are also 'width aware'.

3.4 Pointer-model in action

To give an overview of the changes to the model discussed in this chapter, this section will give a more sizable example. Consider code snippet 3.7, which contains some very precarious pointer tricks. For this example, it is assumed that the smallest addressable data unit for the target architecture is a byte. The translation of the code is shown in figure 3.8a.

Code snippet 3.7 Typed arithmetic on pointers

```
{
  int a[2] = {1, 255}, *p = a, c = 255;
  (*p++)++;
  *(((char *) p) + 2) += (char) c;
  ...X...
}
```

Step 1

The very first FETCH in the block corresponding to line 2 of the code can be transformed over the *Store* and CREATE of `c` and can than be deleted, because it now immediately follows a *Store* on the same address. When deleting a FETCH like this, its output should be connected to the data input of the corresponding *Store*. Looking at said input it should be observed that the `(int *)` cast of the original array pointer `a` is a constant operation on a constant input and can thus be propagated. Since integers have a width of four bytes, the type field of the pointer after the cast should be 4, making the complete pointer $\langle 1, 0, 4\rangle$.

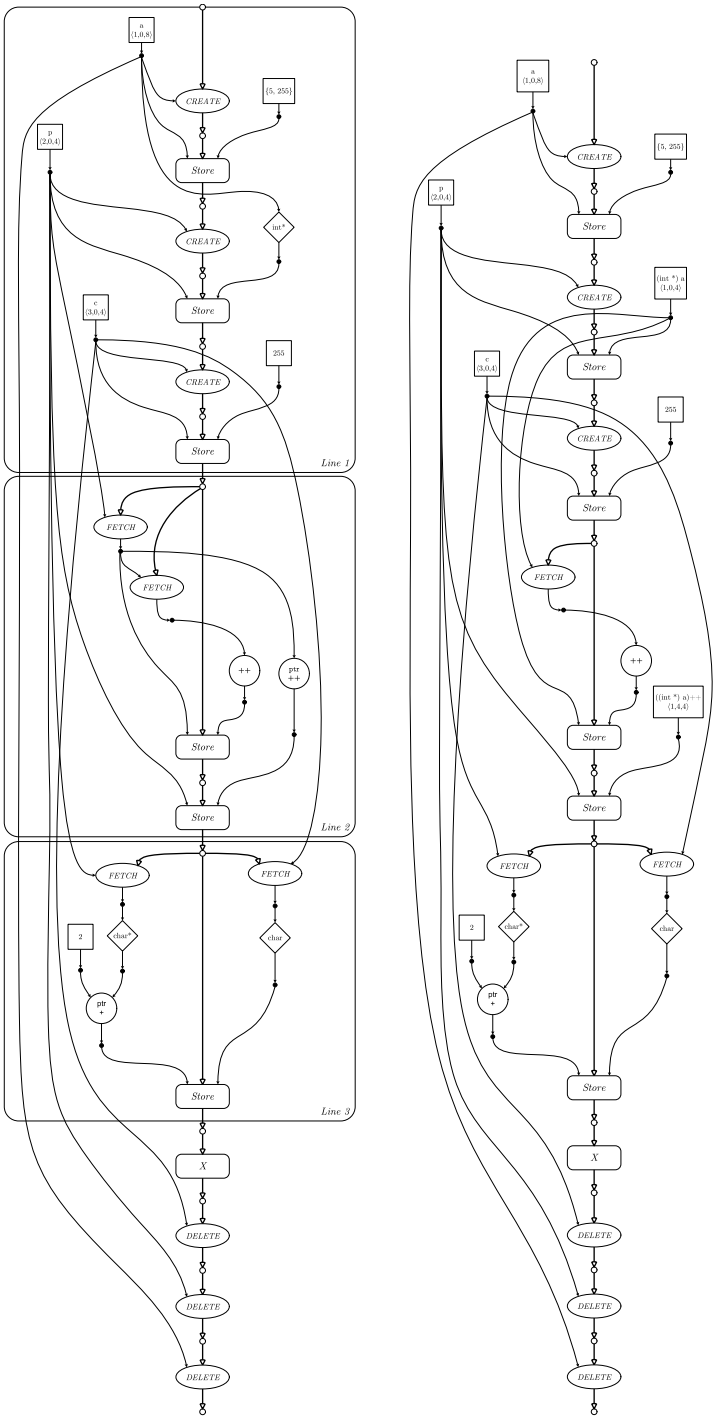
One of the edges taking the resulting pointer as input is the pointer incrementation. This operator 'increments' the pointer, not by simply adding 1 as is the case with integers and other incrementable types, but by the type argument, i.e.

$$\langle 1, 0 + 1 \cdot 4, 4\rangle = \langle 1, 4, 4\rangle$$

These transformations result in the graph shown in figure 3.8b.

Step 2

In the updated situation, the FETCH connected to the `(int *)` cast of `a` (propagated in the previous step), can be transformed upwards, past the *Store* and CREATE of `c` and past the



(a) Translation of snippet 3.7. (b) After step 1.

Figure 3.8: Initial graph and the graph after first transformations

Store and CREATE of p, because even though an address with allocation id 1 is *stored on* p, p *itself* has allocation id 2 and thus the two operation are not interdependent. This brings the FETCH operation being transformed immediately next to the corresponding *Store*. Even though the *Store* of a ($\langle 1, 0, 8 \rangle$) and the FETCH of `(int *) a` ($\langle 1, 0, 4 \rangle$) operate on different sizes, it is possible to replace the FETCH with a constant based on the data input of the *Store*. The resulting constant 5 can also be propagated through the connected increment operator.

Another FETCH that can be transformed out of the graph is the one performed on p, which is an immediate successor of a *Store* on the same address. It can thus be replaced by a direct connection to the data input of the *Store*. This results in a nicely propagable subgraph of exclusively constant edges. The cast to a character pointer results in $\langle 1, 4, 2 \rangle$ and the following addition of constant 2 results (fig. 3.9a) in

$$\langle 1, 4 + 2 \cdot 1, 1 \rangle = \langle 1, 6, 2 \rangle$$

Step 3

The last remaining FETCH in the graph (of c) can be transformed upwards, similar to the ones before. This connects the constant 255 to the cast operator. This is a propagable constant operation and can thus be replaced by a constant. Since data casts only affect the width of the data, a subscript is added[4] to indicate the width in bytes.

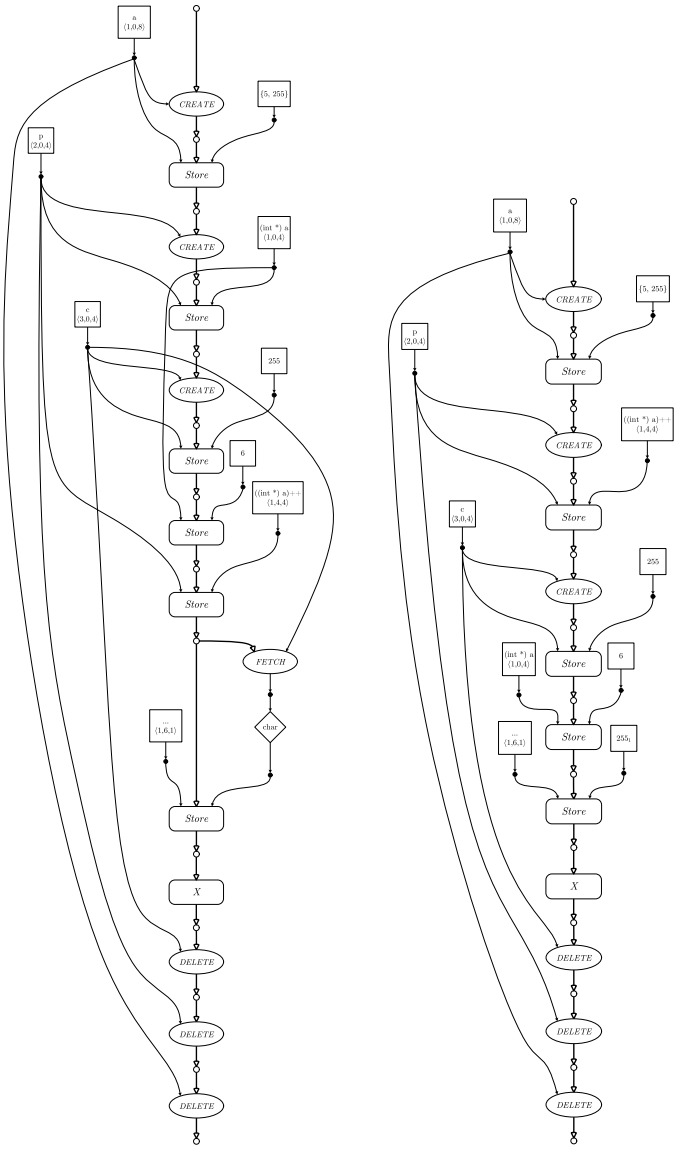
Because all FETCH operations have been transformed out of the graph at this point, the second *Store* of p can be transformed upward across the operations on `(int *) a` and c (as neither have allocation id 2). After this transformation the *Store* is the immediate successor of another store on p and thus overwrites the latter's effects. The first can thus be discarded, resulting in 3.9b.

Step 4

The remaining transformations are those that carry the two bottom *Store* operations upward to their corresponding CREATE. They are both operations on allocation id 1, which is that of a. They can both be transformed without any difficulty to the initial *Store* of a. Since alignment of arrays is fixed in c++, the *Store* on $\langle 1, 0, 4 \rangle$ can be propagated and combined with the initial *Store*, resulting in a single operation. The second *Store* being transformed upward can *not* be unified, because no assumptions have been made so far about the architecture with respect to endianess and thus the alignment of characters in the space of an integer is unknown.

The final result is shown in figure 3.10. If at this point the designer can indicate the target architecture is big-endian the double *Store* operation at the top of the graph can be unified to a single store, storing $\{6, 65535\}$. The translation back to c++ of the graph is shown in snippet 3.8 together with the result if the designer would indeed perform the final transformation assuming big-endianness of the target architecture.

[4] Subscripting the width of a constant is an ad hoc choice made here and different implementations may have different ways visualizing this



(a) After step 2. (b) After step 3.

Figure 3.9: Intermediate transformation results

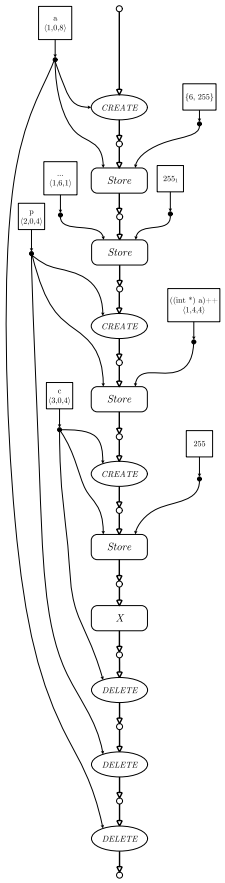


Figure 3.10: Final result after transformations

Code snippet 3.8 Typed arithmetic on pointers

```
{
  int a[2] = {6, 255};
  *(((char *) a) + 2) =
              (char) 255;
  int *p = a[1], c = 255;
  ...X...
}
```

be ⇒

```
{
  int a[2] = {6, 65535},
      *p = a[1], c = 255;
  ...X...
}
```

Summary

Pointers should be transformable through pointer arithmetic, but should *not* cross the boundaries of the memory blocks into which they point. To guarantee pointers can not traverse into coincidentally neighboring blocks of memory, all allocations are modeled as strictly unique, using an allocation identifier that is guaranteed to have a one-to-one relationship with the block of memory resulting from the allocation.

Under the uniqueness constraint, all transformations are legal. They are modeled using transformations on the offset from the beginning of the allocated block of memory. These offsets are expressed in terms of the smallest addressable entity of the architecture (or target language) and *not* in terms of elements of the associated type.

Types are relevant at run-time only for data alignment and are thus only needed when having to extract data from the heap. Therefor, addresses carry the type (width) of the data *pointed to* and data has no need for explicit typing, since it implicitly describes its type by having an unambiguous width.

The final (formal) specification of pointers (addresses) can be found in section 5.1.3.

4 Jumps

This chapter discusses the differences in jump scenarios (4.1) and all these different scenarios are reduced to the generalized form. Next the machine behavior of jumps is analyzed (4.2) and an attempt is made to identify the theoretical minimum of information required to model jumps. Something needs to be said about scoping in the context of jumps (4.3), before finally giving a general model for all classes of jumps (4.4).

4.1 Selected notes on types of jumps

Essentially jumping constructs can be categorized as ‘structural’ and ‘unconditional’. A construct is considered structural, when it delimits the full block of code that the jump crosses. Selection and iteration statements constitute the structural jump constructs in c++. All these structural constructs are mapable to unconditional constructs and—in particular—to the ‘most unconditional’ jump, i.e. the `goto`. These categories of jumps will be treated separately in the next few pages.

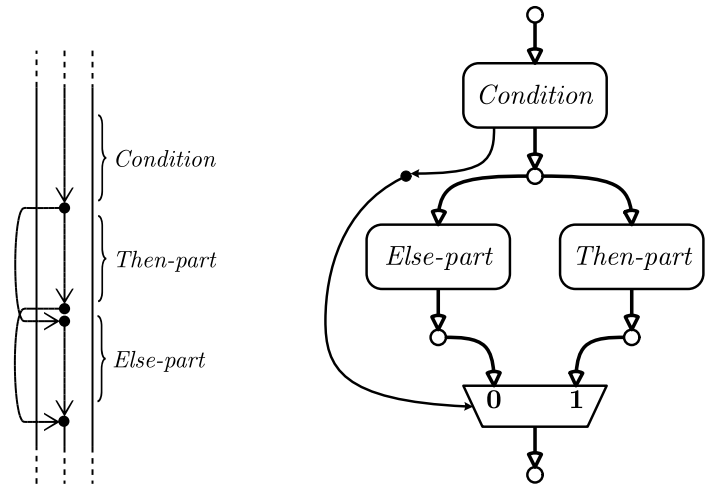
4.1.1 Selection statements

Selection statements redirect the flow of control through code block alternatives, i.e. segments of code are either selected for execution or passed by. The selection statements in c++ are

`if` *condition*
  statement
`else`
  statement

and

`switch (`*condition*`)`
  statement



(a) Linear machine model (b) (Code)block model

Figure 4.1: Selection in the linear and block models

These statements have possible side effects in their conditions, so these should first be allowed to alter the statespace before any block is executed.

It actually depends on the model chosen whether these selection statements constitute jumps or not. In the linear machine model, they do (fig. 4.1a), but in a (code)block model (i.e. SILC) there really is a notion of selection (fig. 4.1b), hence the name.

Selection statements add a single level of scoping. Anything declared in the *condition* is in scope in the nested *statement*(s), but out of scope outside of the construct. Implications of this observation will be treated in section 4.3.

It deserves mentioning that `switch`-statements use `case`-labels only as jump targets. These labels do not alter the flow of control in any way [5, 6.4.2] and the target is chosen beforehand. This is why case labels have to be compile-time constants and can not alter the statespace.

Code snippet 4.1 Case label illustration

```
int i = 2;
switch(i)
{
case 1:
  ...A...
  break;
default:
  ...B...
  break;
case 2:
  ...C...
}
```

As an example, see code snippet 4.1. Only code block *C* will be executed in this example. The occurrence of `default` *before* the appropriate case does not make a

difference, because the target is chosen at the top. This means that the compiler would have to gather information from the nested statement to find out what jump labels are actually available, as opposed to the per-keyword translation that can be performed on an `if`-statement. This is made even more complex by the fact that case labels are not constrained to the current scope, they may well cause jumps *into deeper scopes*.

4.1.2 Iteration statements

The c++-standard [5, 6.5/1] specifies three iteration statements[1], viz.

`while (` *condition* `)`
  statement

`do`
  statement
`while (` *expression* `) ;`

`for (` *for-init-statement* *condition*$_{opt}$ `;` *expression*$_{opt}$ `)`
  statement

Code snippet 4.2 Invariant of a `for`-loop as a `while`-loop

```
for (I C; E)          }  ≡  {
  S                          I
                             while (C)
                             {
                               S
                               E;
                             }
                           }
```

The last two of these have very well known invariants in terms of the first (and vice versa). A `for`-loop can be rewritten to a `while`-loop as shown in snippet 4.2 (note that the braces *are* significant for scoping). Snippet 4.3 shows the reverse mapping.

Code snippet 4.3 Invariant of a `while`-loop as a `for`-loop

```
while (C)          }  ≡  for (;C;)
  S                          S
```

The `while`-statement (and thus any iteration statement) is simply a very special case of recursion [15]. In c++, however, there are some specifics with respect to scoping. Iteration statements—like selection statements—add one level of scoping, but it is important to note that every iteration jumps to a point *before the entry of said scope*. The consequences of this will be discussed in section 4.3.

[1] The *for-init-statement* ends with its own semicolon.

4.1.3 Unconditional jumps

There are a few different types of unconditional jumps. They are

```
break;
continue;
return [expression];
goto identifier;
```

The `break`-statement is used to jump out of iteration statements and `switch`-statements (there would be no use to use it to jump out of `if`-statements, because it is usually used within an `if`-statement inside an iteration or `switch`-statement. The `continue`-statement can only occur inside iteration statements and it jumps to the 'end of the current iteration'.

Both of these statements can be rewritten to `goto`-statements, albeit that some label-generation (and guaranteed uniqueness) is required. In a sense, one could say that an iteration statement binds all free occurrences of `break`- and `continue`-statements (see [1], 6.2.2) and that these jump statements may not occur globally unbound.

Because of this binding requirement, it could be argued that `goto`-statements are 'more unconditional,' i.e. their application is also unconditional as opposed to that of `continue`s en `break`s. The only restriction on a `goto` is that it has to jump to a label *inside* the current function.

Only the `return`-statement is not strictly rewritable to a `goto`-statement, because it explicitly exists the current function scope. However, modelwise a function can be defined as having a `__return_value__` variable by default of the same type as the return type of the function itself. It would then 'return' (i.e. result in, evaluate to) whatever value is stored in said variable whenever it comes to its end.

All these jumps are unified to their `goto` equivalents in the next section.

4.1.4 Unified jumping

As a matter of fact, all these different types of jumps are unifiable in a single jumping model, using only `goto`s and stripped down `if`s. Basically, the unified jumping model is very closely related to stack machine models. In this model `if`s are only allowed to have `goto`s in their bodies and never have an `else`. This closely models the notion of 'jump if nonzero' and its antonym when using the `!` (not) operator.

In the following loose translations conditions `C` will be assumed to have a declaration of type `T` and `C'` will be said condition without the declaration[2], i.e.

```
C ≡ T c = C'
```

When conditions do not contain such a declaration, their translation can be the same, only with omission of the explicit declarations given below.

A translation of the 'normal `if`' now looks like snippet 4.4. The `switch`-statement is a bit more complex because the alternatives given in the case labels must be gathered in a separate pass by the compiler first. The translation then follows as shown in snippet 4.5. When a `switch`-statement does not have a `default`-label in its nested statement, `goto default;` is changed to `goto brk;` (or the `default`-label is inserted right before the `brk`-label. Any free occurrence of `break`-statements in `B0` through `Bn` are substituted by `goto brk;`.

[2] It should be understood that multiple declarations are possible in the condition, but for brevity, a single declaration is assumed here.

Code snippet 4.4 Translation of the 'normal `if`'

```
if(C)
 S1
else
 S2
```

≡

```
{
  T c = C';
  if(!c) goto else_part;
  S1
  goto end_if;
else_part:;
  S2
end_if:;
}
```

Code snippet 4.5 Translation of the `switch` statement

```
switch(C)
{
  B0
case X1:
  B1
case X2:
  B2
  ...
default:
  Bn
}
```

≡

```
{
  T c = C'
  if (c == X1) goto case_X1;
  if (c == X2) goto case_X2;
  ...
  goto default;
  B0
case_X1:;
  B1
case_X2:;
  B2
  ...
default:;
  Bn
}
brk:;
```

Iteration statements can also be rewritten to the unified model. Firstly, the `do-while` statement translates directly (snip. 4.6). Similarly, the `while` statement can be translated (snip. 4.7). Since the `for` is just an invariant of the while—as discussed before—the translation shown in snippet 4.8 follows intuitively. In this last translation, the nested `while` should be translated according to the `while` translation shown before.

Similar to the `switch`-statement translation, all free occurrences of `break` are translated to `goto brk;` and likewise free `continue`s are translated to `goto continu;`. It goes without saying that these labels require some administration to keep them unique, but for legibility they have been kept simple here.

Finally the `return`-statement should be translated to this unified jumping model. The problem that arises, though, is that `goto` is defined to jump to a label *in the same function*. As mentioned earlier reformulating the call/return model solves this problem: Say every function has—per default—a declaration of a variable with the same return type as the function called, the contents of which is handed back to the caller after the function finishes. For a given value `x` of type `T` the `return`-statement can now be translated as shown in snippet 4.9.

Code snippet 4.6 Translation of the `do-while` statement

```
do                          }     {   begin_loop:;
  S                         }  ≡  {   {
while(C);                   }     {     S
                                  {   continu:;
                                  {     if(C) goto begin_loop;
                                  {   }
                                  {   brk:;
```

Code snippet 4.7 Translation of the `while` statement

```
while(T c = C)              }     {   begin_loop:;
  S                         }  ≡  {   {
                                  {     T c = C'
                                  {     if(!c) goto brk;
                                  {     S
                                  {   continu:;
                                  {     goto begin_loop;
                                  {   }
                                  {   brk:;
```

4.2 Jumps in theory and practice

Modelwise jumps are the transferal of control to code at the point jumped to. This point is a location 'in the program', i.e. the program code in memory. In the heydays of flowcharts this translated simply to an arrow pointing to the element corresponding to the instruction found at said code location. However, this 'transferal of control' requires a more fine grained exploration.

In real world architectures jumps correspond either to writing the memory address into some form of current code register (or program counter), or a relative transformation is performed on such a register. If the state of a machine is taken to be the collection of all values stored in its registers, then it is implied that jumping transforms the machine state.

The question is, though, whether the 'location in the code' should be such an *intrinsic* part of the state, or whether the collection of all data and whether or not currently jumping defines the intrinsic state of the machine and the location in the code is an

Code snippet 4.8 Translation of the `for` statement

```
for(I C; E)                 }     {   {
  S                         }  ≡  {     I
                                  {     while(C)
                                  {     {
                                  {       S
                                  {       E;
                                  {     }
                                  {   }
```

Code snippet 4.9 Translation of the `return` statement

```
T f(...)
{
  ...
  return x;
  ...
}
```

≡

```
T f(...)
{
  T __return_value__;
  ...
  {
    __return_value__ = x;
    goto endfunction;
  }
  ...
endfunction:;
}
```

extrinsic parameter. As a matter of fact, this view could be reformulated by stating that the program and the set of primitives and definitions used *in* the program, together define the machine. When formulated as such, the data at a certain time defines the state at that time. Time itself is expressed in terms of the location in the code.

If such is the case, modeling iterations and backward jumps as loops to "earlier" points in the code would indicate revisiting past times, which is impossible. Hence, backward jumps need to be modeled by forward jumps *into a recursion*. These forward jumps have the same semantics as continuations [13]. Hence, every location in the code (where recursion is expanded) is only ever visited once, thereby indeed uniquely modeling time *and* allowing for a complete deduction of the statespace up to that location, by means of the location itself. A problem that remains in modeling c++ this way is the declaration and destruction of variables. This will be discussed in the next section.

4.3 Jumping scopes

C++ permits jumping into and out of scopes of variables. It is even possible to jump from a scope to another (disjunctive) scope at the same hierarchic level. Programmers are, for example, allowed to create programs that jump from the then-part to the else-part of an `if`, although very few of their colleagues award bonus points for style when they do this[3].

4.3.1 Jumping into scopes

Whenever a variable is in scope, its existence is guaranteed, so when jumping over a declaration, but into its scope, the variable should exist, i.e. the declaration should take place. Since the c++-standard specifies that jumping over a *declaration with initializer* of a variable is illegal, it could be argued that declaration is exclusively compile-time. The set of operators that handles declaration and destruction of variables in a model thus forms a different class than that of the run-time operators. Different classes can be defined to have different behavior under the same circumstances, so it could well be argued that the class in which the scoping operators are contained is defined to be

[3] Admittedly, some do [14]

Code snippet 4.10 Illegal jump across declaration *with initializer*

```
...
goto X;
...
{
  int i = 0;
  ...
X:i = 1;
  ...
}
```

independent of whether or not the current state indicates a jump being performed, while other operators are defined under the condition a jump is *not* being performed.

Code snippet 4.11 Legal jump across declaration followed by assignment

```
...
goto X;
...
{
  int i;
  i = 0;
  ...
X:i = 1;
  ...
}
```

Consider as an example snippet 4.10. The c++-standard specifies that this program is ill-formed ([5], 6.7/3), because `i` is declared *with* an initializer. The program in snippet 4.11 is *not* ill-formed, because `i` is declared *without* an initializer. The fact that in the second example `i` is initialized in the assignment on the line right after the declaration is not a problem. A programmer must simply accept and be aware of the fact that `i`'s value is undefined at the point of entry when jumping to `X`. Therefor, a piece of code such as shown in snippet 4.12 is well-formed, but unwise!

Code snippet 4.12 Undefined behavior for variable `i` and thus for `f`

```
...
goto X;
...
{
  int i;
  ...
X:f(i);
  ...
}
```

4.3.2 Jumping out of scopes

The opposite of the above is true also. When jumping out of the scope of a variable its existence ends. Consider snippet 4.13 where the jump to `L` causes the destruction of variable `i`.

Code snippet 4.13 Jumping out of the scope of `i`, which is thus destroyed

```
...
{
  int i;
  ...
  goto L;
  ...
}
...
L:;
...;
```

This mechanism can be modeled, making the same observation made above with respect to the model's scoping operators. A variable's scope ends when the (possibly composite) statement ends, which usually comes down to the first closing brace not bound in the same context as the variable. This seems rather obvious, but the beginning of a variable's scope might be slightly less obvious. Since c discriminates between declarations and statements, a variable's scope always cover the complete statement list. In c++ (and c99 [6]) statements and declarations are mixed and thus the scope of a variable *has* to begin at its declaration.

The beginning of a variable's scope is relevant when jumping backwards, because in that case the variable needs to be destroyed *before* it is recreated by passing the (re)declaration. In the case of Plain Old Data (POD [5, 3.9]) this behavior can be trivially modeled, but in the case of object destruction the object's destructor is called *as if not jumping at all!*

Recall now the translations of the iterative statements in section 4.1.4. A consequence of the rules around jumping scopes is that variables created in the nested statements, or *even in the conditions* are destroyed and recreated upon each iteration.

4.4 Revised jumping model

Based on the observations in the sections above, a new way to model jumps should be formulated. To get a structural perspective, all jumping scenarios are recapitulated (4.4.1), which will result in a singular model for jumps (4.4.2). Lastly, the proof of equivalence of these new templates with the older ones under the rules of transformation is given (4.4.5).

4.4.1 Recapitulating: jumping, scoping and recursion

All legal jump scenarios are shown in snippet 4.14. They can be divided in two groups by their direction (forwards and backwards) and both of these groups can be subdivided by their scoping span: common (staying in scope), into, 'outof' and over. Table 4.1 shows which jumps from the snippet are of which class.

Code snippet 4.14 All legal jump combinations

```
...
goto X;
...
A:
...
{
  ...
  B:;
  ...
  int i;
  ...
  C:;
  ...
  goto Y;
  ...
  D:;
  ...
}
...
E:;
...
goto Z;
...
```

subclass	Forward	Backward
common	X=A X=B Y=D	Y=C Z=B Z=E
into	X=C X=D	Z=C Z=D
outof	Y=E	Y=A Y=B
over	X=E	Z=A

Table 4.1: Classification of jumps (see snippet 4.14)

Forward jumps impose little implications, they merely cancel the evaluations of all following *run-time* operators up until the point indicated by the label jumped to. By imposing on the jump model that all *compile-time* operators *are* evaluated when jumping, the creation and deletion of variables is guaranteed and thus jumping forward into, out of and over scopes follows naturally.

Backward jumps are harder to model since cycles are not allowed in the graph. Hence, backward jumps do not exist as such in the model and should be modeled as forward jumps into a recursion of the scope. Recursion *within* the scope is important, since the compile-time operators are evaluated and the scope can not be exited and entered without loosing values assigned to the variables of said scope. This idea covers the common backwards jump intuitively, but it also covers the into and over cases, because jumping backwards into a nested scope, means the jump is into a scope contained in the current one. The same holds for jumping backwards over scopes, or rather, a backwards jump over a scope is a forwards jump to a point before the beginning of the scope (in the recursion) that it is said to jump over.

This leaves one unhandled case: the backward jump out of the current scope. Al-

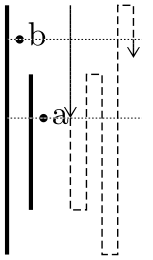


Figure 4.2: Control flow when jumping backwards out of the current scope

though, the scope jumped into[4] has its own recursion construction and will thus provide a means to come across the target label when jumping into it. Figure 4.2 shows a backwards outof jump from *A* to *B*, where the solid line is the path of execution and the dotted line the path of the label search (in the model, this too is 'execution', but excluding run-time operators). Therefor, the only real unmatched case is that of a jump out of the top-level scope. Recall from sections 4.1.3 and 4.1.4 that `goto`s are only allowed from one point within a function to another and that a function always has the implicit definition of `__return_value__`. Since the definition of this return value variable is implicitly contained in the function definition itself, the top-level scope in that function is always nested in the scope of the return value variable's scope.

4.4.2 Scope-safe jumping

In order to be able to formulate a new model, a few operators will be introduced based mainly on intuition here for brevity. Their formal definition can be found in chapter 5.

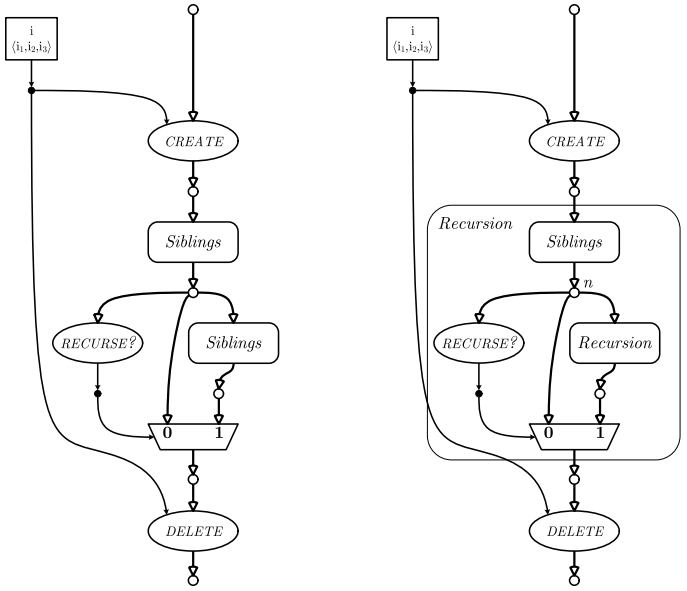
For the most basic modeling of jumps a 'jump' operator and an accompanying 'land' operator—both with the jump label as an argument—are required, but in order to determine the complete functionality of these operators the remainder of the jumping model must first be observed. As shown in section 4.4.1, recursion is required to facilitate backward jumping. First off, it is important to know how much of the program the recursion should contain. Intuitively this is all the code in the scope, but there must be stricter definition.

Assuming that every statement is part of some compound statement[5], it is reasonable to say that all statements between a declaration and the end of the compound statement are in scope of said declaration. The statements after a declaration up to the end of the compound are hereafter considered the declaration's siblings.

Figure 4.3a shows a possible modeling of the ideas laid out above. Since recursion should only occur when the *Siblings* edge is exited with a jumping state, it should not be an unconditional successor. Supposing there is some operator that can deduce from the state whether or not a recursion is required, the recursive branch should either be selected or ignored. If the jump performed in *Siblings* is a backward common jump, the corresponding land operator will be found in *Siblings*2, but it is possible that the evaluation of the latter will result in the same sort of jump. When this occurs, this model does not suffice, since there is no recursion after *Siblings*2. As a matter of fact, what

[4]This must be on a higher level than the current, since the jump is outof.

[5]This assumption is perfectly sound, because at the top-level of a program there are only variable declarations with global scope and function definitions. The function definitions have compound statements in which all actual execution occurs.



(a) First attempt at recursion (b) Real recursion

Figure 4.3: First two steps on the way to a jump model

is depicted here can not be considered true recursion, since the recursion mechanism is not included in itself.

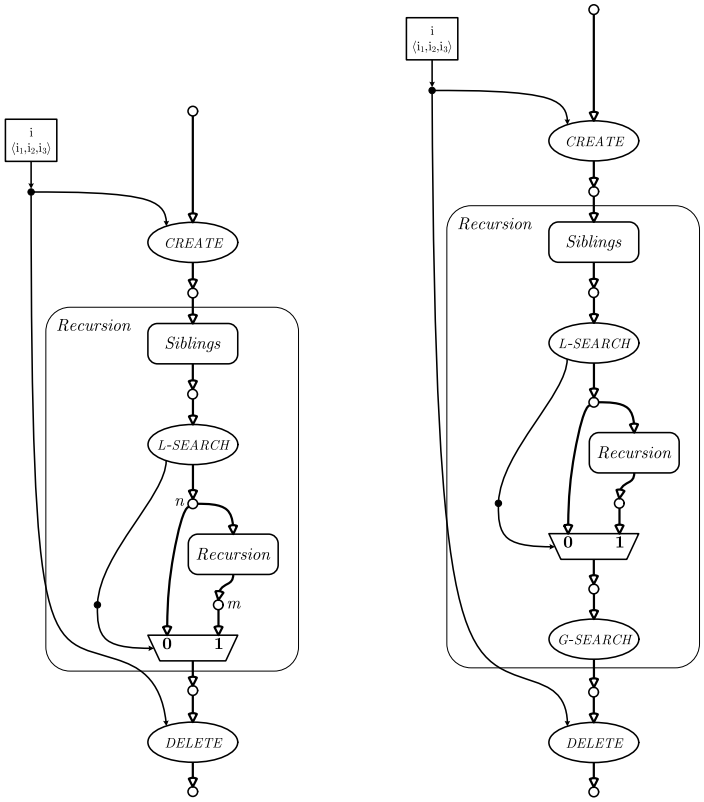
The model depicted in figure 4.3b remedies this. However, it falls short with respect to forward outof jumps. Since the *Recurse?* operator only reads the state (and does not alter it), the state that determines the necessity of recursion (at node *n*) is the state passed into *R*. In the case of a forward outof jump, the corresponding land operator will not be found in *R.Siblings*[6] and thus the state at *R.n* will be identical to the state at *n*. At *R.n* the same *Recurse?* operator will then evaluate to true and thus result in infinite recursion.

To prevent infinite recursion the state at node *n* should be altered so that recursion is performed at most once per jump per scope. This implies the recursion test should alter the state to indicate recursion is no longer required. Figure 4.4a shows a new operator—named *L-Search* to indicate that it searches the local scope for the corresponding land operation—that evaluates to both an indiction whether or not a recursion is required *and* a state in which it is guaranteed that no (subsequent) recursion is requested. This model still shows one deficit: backward outof jumps do not behave correctly. When the target label is not in recursion *R*, the state at *m* will be equal to that at *n*, so when control is handed back to the outer scope (not in the figure, past the DELETE) the state is set to not recurse, so the *L-Search* of that outer scope evaluates to false and thus the recursion in the outer scope is not selected.

Finally, figure 4.4b shows a model where this flaw is corrected. The *G-Search* operation resets the state to requiring recursion when coming across an *L-Search* operation. This model does cover all classes of jumps shown in table 4.1, the reader is encouraged to check this.

Summarizing, the operations *L-Search* and a *G-Search* suffice to make a jump

[6] The *Siblings* edge *in R*.



(a) Locally correct attempt (b) Globally correct attempt

Figure 4.4: Last two steps on the way to a jump model

model scope-safe. What remains to be discussed is a more precise correspondence between these operations and the jump and land alternatives. What is given so far is that these are transactions on the statespace and that both the target-label and some indication of whether or not recursion in the current scope is required. What has not yet been discussed is what labels are in terms of the model.

4.4.3 Transformations on jumps and identity of labels

As shown in chapter 2, transformations can be seen as offline evaluation. Since the whole purpose of SILC is to create an easily optimizable structure from a program, the operators chosen to model jumps should be optimizable. First of all, some correspondence with the search operators from the last section is required. This implies that there should be a way to indicate to the jump operator that the jump is either in the local scope or out of it (or at least, whether or not to search in the local scope for a label). Next, chosen operators should be transformable without altering the external behavior of the program.

Consider the situation in figure 4.5, which will occur (after transformations) quite commonly. If $L_1 = L_2$, it means that the jump has reached its destination and should thus end. The naive transformation would remove *both* the jump and the land operators, but if there is a jump operator corresponding to this land operator prior to this local

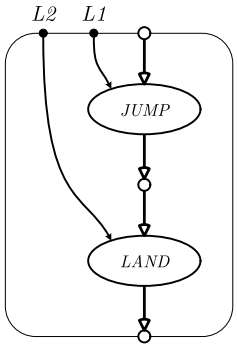


Figure 4.5: Possible transformation problem

situation, the land operator is still required. Hence, the proper transformation only removes the jump operation. As will become apparent in section 4.4.5 this will—in real life examples—inevitably lead to situations where $L_1 \neq L_2$, in which case neither can be removed and their order can not be reversed. This seems to be fully consistent with desired behavior, but because of the afformentioned transformation when $L_1 = L_2$ it might very well occur that there is no longer a corresponding jump operation for the land operation. When this happens, said land operation is a placeholder that has lost its semantic value.

In this case there should be some way of telling that there is no such corresponding jump operation, i.e. labels should also have a point of introduction before which they are guaranteed not to be used and a point of destruction, after which the same guarantee can be given. Basically, this means labels—like variables—have scopes. When looking at what labels *are* in the machine context, it turns out that they are addresses. Of course this is because of the fact that code and data are both in memory, but it works just as well to model what labels are, especially since any newly created address is guaranteed to be unique. Beyond this point, labels will be modeled the same way as variables of a zero-length type.

Labels will thus be introduced by a CREATE and destroyed by a DELETE. The problem shown in figure 4.5 can now be solved by transforming the CREATE of L_2 downward over the jump operation shown here. When a CREATE is immediately followed by a land operation with the same argument, said land operation can be deleted, because it is guaranteed that there is no jump to it. This means the representation of labels in the statespace is fixed. Representations for the recursion control and the indication of what label is 'active' (if any) remain to be chosen.

Both these remaining parts of the state can be modeled by putting them *in* the statespace. There should be some predetermined location to store the active label and the recursion control. Since jumping only takes place within functions, it is reasonable to have CREATES and DELETES for these at the beginning and end of every function. As a convention, the addresses of these values will be called $\mathfrak{L}$ (for labels) and $\mathfrak{R}$ (for recursion control).

With these definitions, it is possible to define the jump and land operations. A jump operation should have as its two arguments the label it jumps to and whether or not it has to recurse in the local scope (since every evaluation should have an equivalent transformation, it should be possible to transform a jump into a recursion where it should no longer demand further recursion). However, since the recursion control argument is

always constant, it seems more optimal from a transformation point of view to define alternative operations for jumps (potentially) into the local scope and (guaranteed) out of it. Furthermore, 'jumping' is a term from the machine context and does not correspond well with the line of thinking in the model. Hence the operations *L-Skip* (into) and *G-Skip* (outof) will be defined, both taking a label as an argument. These names correspond nicely with *L-Search* and *G-Search*. Landing now lost its correspondence with jumping, so the alternatively named operator for it will be *Continue*, also taking only a label as an argument.

4.4.4 Refinement—which operations are primitive?

There are two aggravating observations that can be made about the model thus far. First of all, having different classes of operators (although definable) is not very esthetic and can—at times—be very confusing. Second, if labels and recursion control are kept in the statespace, the jump, land and search operators are simple (combinations of) reads and writes on the statespace.

It should thus be possible to express jumps, landings and searches in terms of primitive operations that read and write from and to the statespace. A complete model can be formulated with just these primitives. The formal specification can be found in chapter 5, but an intuitive definition can already be given here. Assume the existence of the following:

- `R(s, a)` that reads from state `s` at address `a`
- `W(s, a, d)` that writes data `d` to state `s` at address `a`
- `if(c, t, e)` that evaluates condition `c` and results in `t` if said condition is true or `e` otherwise.

Using these informal functions, the non-primitive functions from before can be specified as

```
        Skipping(s)  =  R(s, ℒ) != ℒ
Skip(s, l, recurse)  =  W(W(s, ℒ, l), R, recurse)
        Fetch(s, a)  =  R(s, a)
     Store(s, a, d)  =  if(Skipping(s), s, W(s, a, d))
     Continue(s, l)  =  if(R(s, ℒ) == l, Skip(s, ℒ, -), s)
       L-Skip(s, l)  =  if(Skipping(s), s, Skip(s, l, true)
       G-Skip(s, l)  =  if(Skipping(s), s, Skip(s, l, false)
        L-Search(s)  =  ⟨W(s, R, false), Skipping(s) && R(s, R))⟩
        G-Search(s)  =  W(s, R, true)
```

4.4.5 Proof by transformation

To prove that SILC is compatible with *both* the linear machine model *and* the (code)block model, a translation of the `f` that intuitively relates to the first, will be transformed to the second. Compare the code snippet 4.15 to its translation shown in figure 4.6a. The graph shown is very similar to common machine code translations and thus reflects the linear machine model. Since the block model is only relevant when *not* jumping into or out of the blocks *C*, *S*1 and *S*2, it seems reasonable to assume that operations on labels can be transformed over them (the logic of this will become clear during transformation).

Code snippet 4.15 A common `if` statement

```
if (...C...)
{
 ...S1...
}
else
{
 ...S2...
}
```

Step 1

First of all, *S*1 can be transformed across the mux, placing a copy of *S*1 in each branch. This means the `0` branch now has an *L-Skip* followed by a complex edge of class *S*1. Having made the assumption *S*1 does not operate on labels, there will not be a *Continue* or any kind of skipping operation anywhere in *S*1. It is thus possible to transform it the edge out of the graph by letting the *L-Skip* operation consume it.

The same process should be repeated with the *L-Skip* under the mux. Again, the copy placed in the `0` branch can be consumed by the *L-Skip* edge already in it, because if there is a jumping state (i.e. there is an active label) *before* the upper *L-Skip* in the branch, the active jump will consume *both L-Skip* edges in the branch. If, on the other hand there is no active jump when entering the branch, the upper *L-Skip* will set an active label and thus the lower *L-Skip* will be disabled still. These transformations result in the graph shown in figure 4.6b.

Step 2

Next, the *Continue* below the mux is transformed upwards. Yet again, branch `0` gives an interesting situation. Now the *L-Skip* in it is the immediate predecessor of the *Continue* transformed upwards before. Both operations take the same label as an argument and thus the *L-Skip* can be transformed out.

The *Continue* in the `1` branch can not be transformed over the *L-Skip* above it, because they take different labels as arguments. To transform it out, the CREATE on the `else_part` label needs to be transformed downward. It can be transformed over *C* under the prevailing assumption that *C* has no operations on labels (fig. 4.7a).

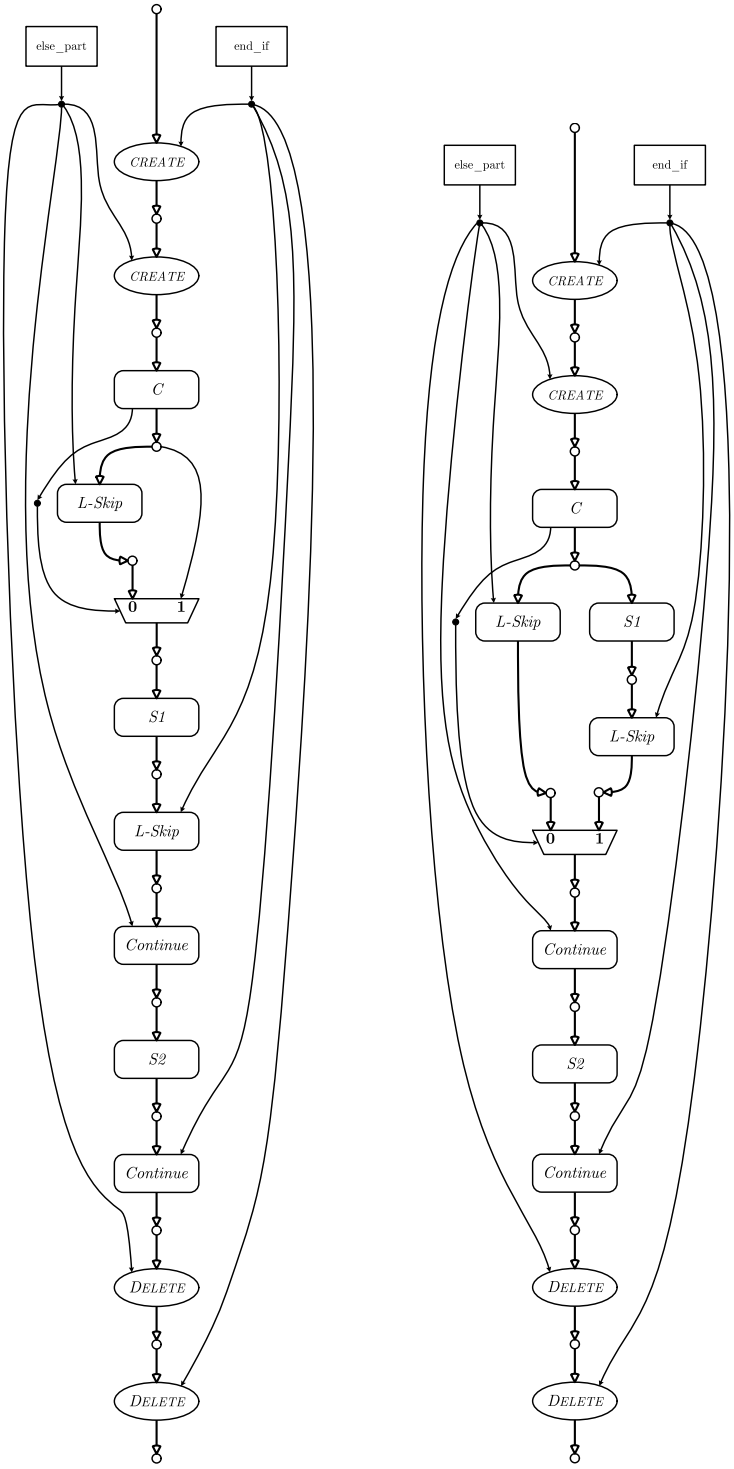
Step 3

The CREATE can be carried into both branches. In the `0` branch, it consumes the *Continue*. In the `1` branch, it can be transformed over *S*1 similar to the transformation across *C* and finally over the *L-Skip* on the `end_if` label, because the argument of the CREATE is the `else_part` label.

At this point it has become the direct predecessor of the *Continue* on the `else_part` label and can thus consume it (fig 4.7b).

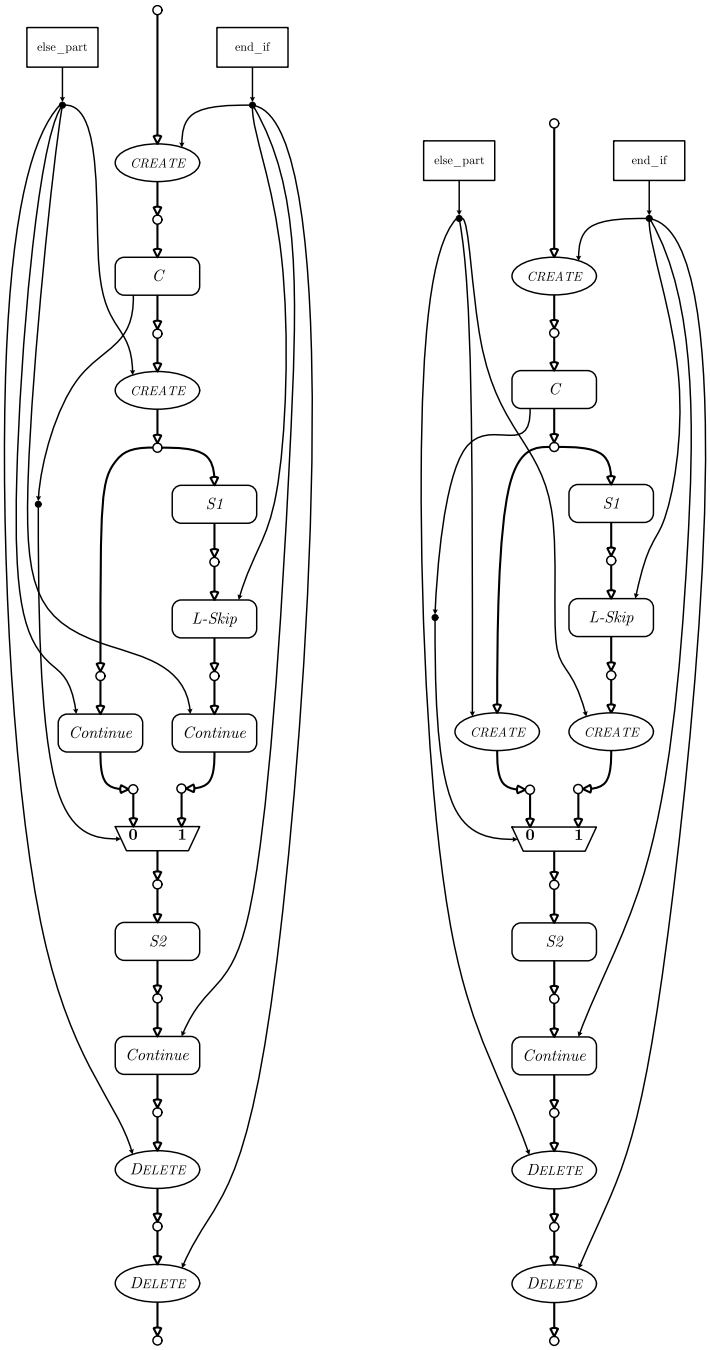
Step 4

Similar to the transformations before, the CREATE of the `else_part` label can be carried downwards across the mux, *S*2 and the *Continue* on the `end_if` label. With the CREATE



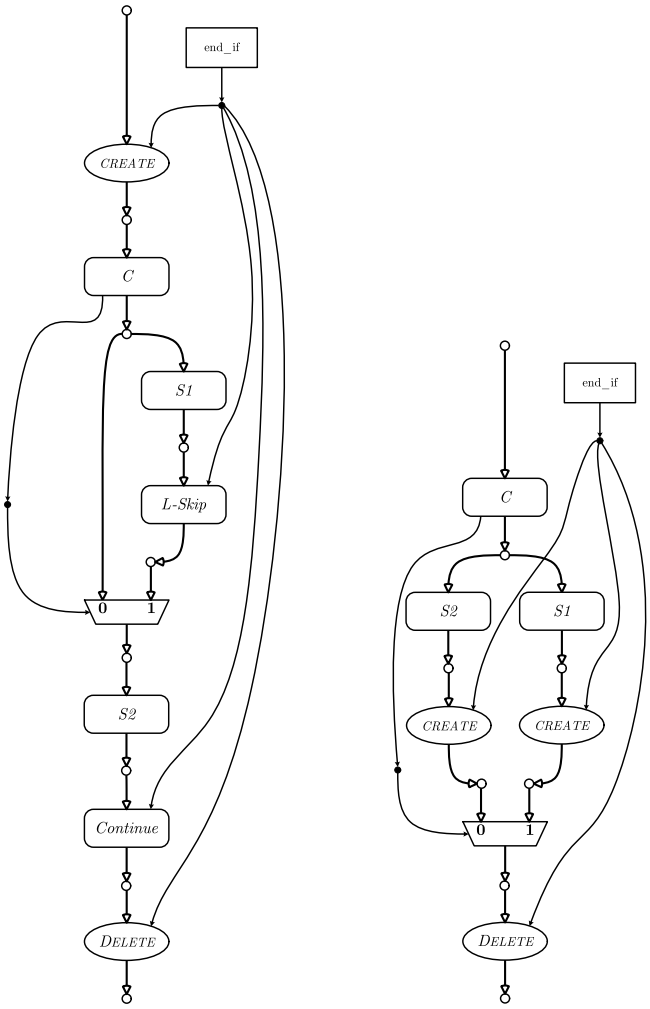
(a) Initial translation (b) After step 1

Figure 4.6: Proof-by-transformation 1



(a) After step 2 (b) After step 3

Figure 4.7: Proof-by-transformation 2



(a) After step 4 (b) After step 5

Figure 4.8: Proof-by-transformation 3

and DELETE—both on `else_part`—in immediate succession, they cancel each other out and can both be removed.

Step 5

Similarly to the previous steps, *S*2 is carried over the mux, consumed by the *L-Skip*in the `1` branch and kept in the `0` branch.

Next the *Continue* is transformed over the mux, where it consumes the *L-Skip* in the `1` branch. Now both *Continue* operations can not be transformed any further, so their corresponding CREATE is transformed down.

It is carried over *C*, into both branches and over *S*2 and *S*1 in the `0` and `1` branches respectively. In both branches the CREATE can now consume the immediately successive *Continue* (fig. 4.8b).

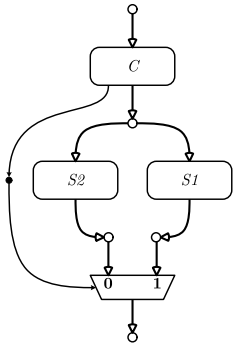


Figure 4.9: Block model resulting from transformation

Step 6

This final step carries the CREATE operations in both branches across the mux, so that a CREATE-DELETE operation pair is again found with identical arguments. They can be transformed out resulting in the original block model (repeated in figure 4.9).

Summary

Because c++ has a lot of behavior hidden away in its jumps, SILC is explicitly designed to unhide said behavior. To do this, all different kinds of jumps are unified to a single kind in the *unified jumping model*, which is simply a rewriting scheme in terms of c++.

A lot of the hidden behavior of c++ occurs when jumping into or out of scopes. As SILC graphs are strictly acyclic, backwards jumps are modeled by forwards jumps into recursions. These recursions are introduced by the introduction of every new level of scope.

Labels jumped to are normal addresses in SILC corresponding to the imperative stack machine world.

It has become apparent that the entire model of jumps can be expressed in the four primitive operations (CREATE, DELETE, STORE and FETCH) already available.

5
SILC, the model

This chapter attempts to describe the SILC-model in a concise and complete manner. First off, the primitive types available in the model are defined (5.1) using the formal specification language Z [7, 12] extended with the toolkit shown in appendix B. Next, the definition of primitive statespace operations are given (5.2), also using Z. Lastly a few commonly used complex operations are described (5.3), which can be considered the standard SILC toolkit.

5.1 Primitives

5.1.1 Data

To maintain generality, the model assumes the existence of an elementary data type that represents the smallest addressable entity in the target architecture. A type is required to describe this unbreakable entity:

$$[\mathfrak{D}]$$

This definition enables the definition of a more global notion of data. Even though SILC theoretically models infinite data sequences, sequences of data are defined as *finite* sequences here, to relate closer to the target architecture.

$$\mathbb{D} == \operatorname{seq} \mathfrak{D}$$

Provided there is a notion of a null-value in $\mathfrak{D}$, let

$$\mathfrak{D}_0 : \mathfrak{D}$$

be said null-value. The equivalent null-sequence of data can now be defined as

$$\mathbb{D}_0 : \mathbb{D}$$
$$\forall i : \operatorname{dom} \mathbb{D}_0 \bullet \mathbb{D}_0(i) = \mathfrak{D}_0$$

allowing it to be as big as required. Possibly, constraints on the size of $\mathbb{D}_0$ can be given later, but from hereon, it will be considered to be 'at least as big as the biggest requirement.'

5.1.2 Types

For data alignment purposes, (course grain) typing is required. Since the *only* purpose of the typing system is alignment, the 'width' of data—in terms of $\mathfrak{D}$—will suffice. However, the width of indirections depends heavily on the target architecture *and* the width of the reference says nothing about the width of the referenced data. Thus, indirections need to be contained in the type

$$\mathbb{T} ::= primitive\langle\langle \mathbb{N} \rangle\rangle \mid reference\langle\langle \mathbb{T} \rangle\rangle$$

There is also a generic dereferencing function to determine the referenced type. Furthermore, the existence of a size function for the target architecture is assumed. This function is typed as

$$\begin{array}{|l} * : \mathbb{T} \longrightarrow \mathbb{T} \\ \# : \mathbb{T} \longrightarrow \mathbb{N} \\ \hline \forall\, t, t' : \mathbb{T} \mid t = reference\, t' \bullet *t = t' \end{array}$$

Any instance of $\mathbb{D}$ carries its type implicitly. Observe that

$$\forall\, d : \mathbb{D} \bullet typeof(d) = primitive\,(\#d)$$

5.1.3 Addresses

In general, addresses are offset table, but given two independent pointers i and j then the variable pointed to by j is unreachable from i by whatever offset, i.e.

$$\nexists o : \mathbb{Z} \bullet i + o = j$$

Therefor, the model for addresses should specify interdependence. All dependent address should share a common denominator and from the imperative context this denominator is the allocation. Let

$$[\mathbb{A}]$$

be the unordered set of allocation identities. A complete address can now be modeled as the triple of allocation identity, the offset thereof and the type of the data pointed to:

$$\mathcal{A} == \mathbb{A} \times \mathbb{N} \times \mathbb{T}$$

Offsetting any address can now be achieved by adding to the offset. Offsets can also be reduced by subtraction, but the resulting offset must, of course, remain positive. Regardless of the offset, addresses are related ($\overline{\overline{\wedge}}$) when their allocation identities are equal.

$$\begin{array}{|l} \_ \,\overline{\overline{\wedge}}\, \_ : \mathcal{A} \leftrightarrow \mathcal{A} \\ \_ + \_ , \_ - \_ : \mathcal{A} \times \mathbb{N} \longrightarrow \mathcal{A} \\ \hline \forall\, a, a' : \mathcal{A} \bullet a \,\overline{\overline{\wedge}}\, a' \Leftrightarrow (\pi_1\, a = \pi_1\, a') \\ \forall\, i : \mathbb{A};\ o, n : \mathbb{N};\ t : \mathbb{T} \bullet \\ \qquad (i, o, t) + n = (i, o + n, t) \wedge \\ \qquad ((i, o, t) - n = (i, o - n, t) \Leftrightarrow o \geq n) \end{array}$$

Finally, it is important to observe that there should exist a bijective mapping from addresses to data (pointers are stored in memory). What that mapping looks like explicitly is not very important (and architecture-dependant), but assuming said mapping the following relation can be defined:

$$\_ \equiv \_ : \mathcal{A} \leftrightarrow \mathbb{D}$$

which just states a certain address relates to a certain data, but abstracts away from the mapping itself. Since said mapping is bijective, this abstraction is plausible.

5.1.4 (Data-)memory, model control and state

Imperative languages are based on a model of mixed memory (control- and data memory), but SILC only models data memory as a type within the model. Control memory is modeled as a mix of the instance of the model itself and the state at any given point in said instance.

Data memory is the relation between allocation identities—*not* addresses—and *untyped* data. This provides a very intuitive view on the independence of separately allocated blocks of memory. Besides the type definition for memory, the *allocation uniqueness criterium* is also required to restrict the model to valid memory only.

$$\mathcal{M} == \mathbb{A} \leftrightarrow \mathbb{D}$$
$$\forall m : \mathcal{M} \bullet \forall e, e' : m \mid e \neq e' \bullet \mathit{first}\, e \neq \mathit{first}\, e'$$

As introduced in chapter 4, there is a default location for the active label. This default location is itself an address:

$$\mathcal{L} : \mathcal{A}$$

As concluded in that same chapter, the statespace can be sufficiently modeled by memory (a mapping from allocation identities to values). In terms of the schema for state:

State
$m : \mathcal{M}$

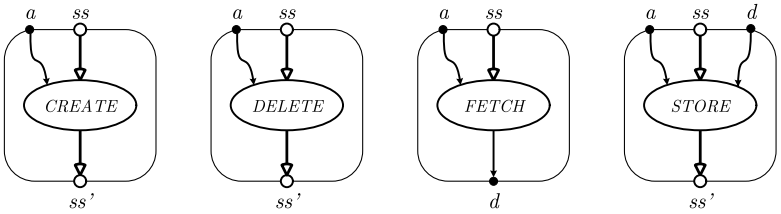
5.2 Primitive operations

Operations on data are heavily target-dependant and can not be assumed to have a useful greatest common denominator. All operations do have the same external appearance; they are all (partial) functions from the values of their input nodes to the values of their output nodes.

The (atomic) operations on the statespace *do* have an architecture-independent nature and can thus be defined here.

5.2.1 Scoping

Variable declarations correspond to allocations in memory. This is modeled directly on statespace operations. Allocations (and deallocations) are always performed on a base address ($\mathbb{A}$), but to maintain some conformity as to the format of addresses,



(a) Allocate (b) Free (c) Read (d) Write

Figure 5.1: Primitive operations

full addresses ($\mathcal{A}$) are used, but required to have a zero offset. Allocation in c++ corresponds to creation (fig. 5.1a) in SILC.

CREATE

$\Delta State$
$a : \mathbb{D}$

$\forall a' : \mathbb{A}; o : \mathbb{N}; t : \mathbb{T} \mid (a', o, t) \equiv a \bullet$
$\quad o = 0 \land m' \setminus m = \{(a', \#t \leftarrow\!\!\square\ \mathbb{D}_0)\}$

Likewise, destruction in c++ corresponds to deletion (fig. 5.1b) in SILC.

DELETE

$\Delta State$
$a : \mathbb{D}$

$\forall a' : \mathbb{A}; o : \mathbb{N}; t : \mathbb{T} \mid (a', o, t) \equiv a \bullet$
$\quad o = 0 \land a' \in \operatorname{dom} m \land$
$\quad m' = \{a'\} \lhd m$

5.2.2 Reading and writing

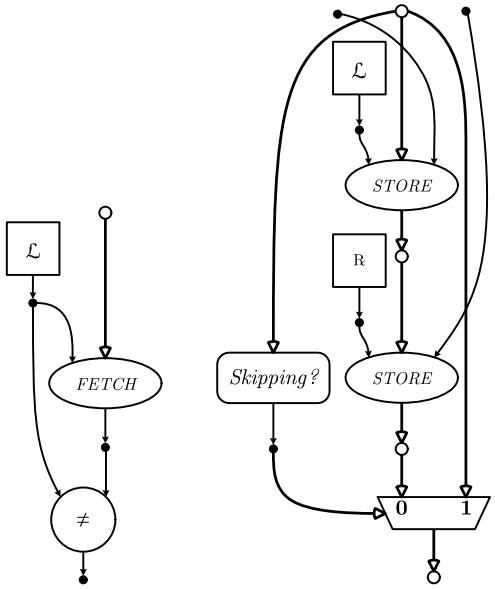
Reading data from the statespace is an operation that only results in the data read, i.e. there is no transformed statespace as a result of a so called FETCH operation. This is modeled explicitly, as one of the intentions of SILC is to unhide side-effects that where hidden in c++. Visually this corresponds to there not being an outward state-arrow (fig. 5.1c).

FETCH

$\Xi State$
$a, d : \mathbb{D}$

$\forall a' : \mathbb{A}; o : \mathbb{N}; t : \mathbb{T} \mid (a', o, t) \equiv a \bullet$
$\quad a' \in \operatorname{dom} m \land d = \#t \leftarrow\!\!\square\ (o \square\!\!\rightarrow m(a'))$

Writing *does*, of course, alter the statespace. The corresponding STORE operation (fig. 5.1d) results in data being written into memory.



(a) *Skipping?* (b) *Skip*

Figure 5.2: Jump testing and activation

STORE

$\Delta State$
$a, d : \mathbb{D}$

$\forall a' : \mathbb{A}; o : \mathbb{N}; t : \mathbb{T} \mid (a', o, t) \equiv a \bullet$
$a' \in \operatorname{dom} m \land m' = m \oplus \{(a', write(o, d, m(a')))\}$

5.3 The SILC toolkit

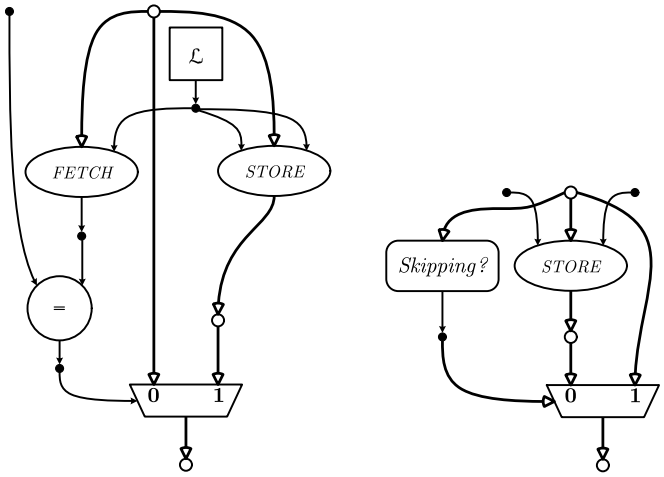
Because some particular arrangements of the primitive operations occur so frequently, a default toolkit is defined.

5.3.1 Jumps

Jumping in c++ is based on labels, which indicate where to jump to. Labels are represented by addresses, and the active label is stored at address $\mathcal{L}$. When there is no active label, the address stored at $\mathcal{L}$ is $\mathcal{L}$ itself. Hence, an operator (*Skipping?*) that tests whether or not a jumps is being performed can be defined (fig. 5.2a, by testing whether the currently active label is *not* $\mathcal{L}$.

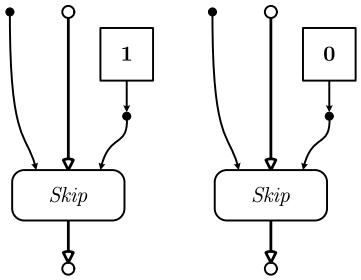
Now the *Skip* operator can be defined, using the *Skipping?* operator, since it should leave the state unaltered if there already is an active label. When *Skipping?* concludes a jump is not being performed, the *Skip* operator should set the active label and the scope recursion control fields. If a jump *is* being performed, the state on the input is copied to the output (fig. 5.2b).

The antonym of *Skip* is *Continue*. Taking as its only argument the label that it represents. If the label on its input is the active label, the jump should end and, thus, $\mathcal{L}$



(a) *Continue* (b) *Store*

Figure 5.3: Landing and jump-safe storage.



(a) *L-Skip* (b) *G-Skip*

Figure 5.4: Shorthands for jumping into and outof scopes

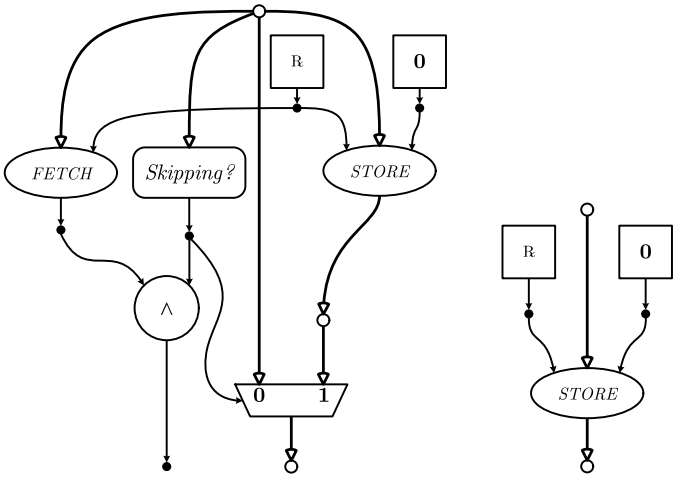
should be made to point to itself again. The definition of *Continue* is shown in figure 5.3a.

5.3.2 Jump-safe storage

The primitive STORE operation writes to memory independent of jumps. Now that the jumping operations have been defined, a higher-level storage operation can be defined that does not write to a statespace when a jump is being performed (i.e. when a label is active). The resulting *Store* operation (5.3b) can be used without having to take further safety precautions.

5.3.3 Searches

Since it is often known what the value of the scope recursion control field should be, default jump operators should be defined to allow for easier transformation definitions (removing the complexity of the extra input). These operators are *L-Skip* for jumps that need to search in the local scope and *G-Skip* for jumps that are outof (see section 4.4). Both of these operations are shown in figure 5.4.



(a) *L-Search* (b) *G-Search*

Figure 5.5: Search operations for scope recursion

The operators that guide the search for the active label (as described in section 4.4.2) are *L-Search* and *G-Search*. The how and why of scope recursion control has been discussed extensively at their introduction, so without further ado, turn to figure 5.5 for their respective definitions.

6 Annotations for optimization

A sizeable number of transformations can be defined on the model as treated in chapter 5. However, the fact that any edge defined by a graph class in the namespace (and not in the default toolkit) requires expansion if other edges are to be transformed over it. Besides the intuitive setback in performance, this might actually result in the inability to perform the desired transformation at transform-time, because it crosses a parameterized recursion—which is thus potentially infinite.

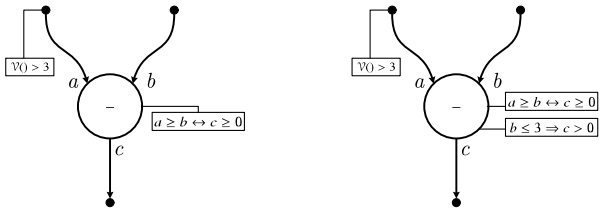
This chapter introduces an optimization by means of annotation to compensate for this shortcoming. The syntax of these annotations is introduced in section 6.1. Section 6.2 shows how these annotations can be used in transformations, followed by the inductive proof of correctness in section 6.3. Finally, section 6.4 shows how these annotations can actually be deduced at compile-time.

6.1 Syntax and the elaboration rule

Annotations are a way to connect assertions to an edge or node. These assertions are properties that follow from the graph itself and thus have no added *semantic* value. There are differences between assertions that can be made about nodes and those about edges. Assertions about edges will, for instance, often indicate (partial) relations on the in- and output nodes or dependencies in terms of addresses when the edge transforms the state, while node-assertions are more value related.

A strict condition for the use of annotations is that no one assertion on a node or edge may contradict another on that same node or edge. The syntax of annotations should allow adding extra information (syntactically) independent of the annotations already made. Consider the notation in figure 6.1a. It shows one annotated edge, one annotated node and two unannotated nodes. Combining both annotations, an extra assertion can be logically deduced about the edge. This deduced assertion is added, by simply adding a new annotation to the edge (fig. 6.1b).

The syntactic independence of these annotations allows for the independent consideration of assertions in transformations (6.2). Any and all annotations are valid at



(a) Annotated graph (b) Propagated annotation

Figure 6.1: Annotation enabling transformation

any given time and thus any one annotation can be used for a more informed pattern match without having to check the other assertions made about an edge or node.

The argument above does assume a system that is guaranteed to retain its consistency even when annotations are added. To provide such a guarantee, the elaboration rule is required, which is defined as follows:

- A set A of annotations is considered *consistent* if it contains no contradictions.
- Elaborating on the properties of a node or edge is possible by adding annotations. A consistent set A can only be extended with annotation ψ if the resulting set $A \cup \{\psi\}$ is consistent.

The assertion made in the annotation should be a logic predicate. To actually realize a means to transform state altering edges over potentially infinitely recursive edges, a way is required to indicate an edge's complete dependencies with respect to given operations. Concretely this implies the requirement for a function that takes as arguments a primitive statespace operation and an edge and returns all the addresses the operation is performed on in said edge. This dependency function will be called Δ and will be written without the edge argument in the annotations, because it is clear from the annotation which edge is its argument. In section 6.4 it will be shown that it is not actually required to give an implementable (i.e. generative) definition of Δ, because all relevant projections can be deduced from compilation.

Since addresses can themselves be stored in memory, an unlimited level of indirection is allowed. This means a notation is required to indicate (the level of) indirection. In Δ-annotations addresses will be given an 'order' by a superscripted number, e.g. b^n is address b in the n^{th} order. The order of an address corresponds to the level of indirection. If data is written to memory at address a, it is a first-order address. If data is written to memory at the address that is stored at address a, it is the second-order of a, i.e. a^2. Snippet 6.1 shows this principle in terms of c++.

Code snippet 6.1 Orders of addresses

```
...   a ... // First-order  address (a^1)
...  *a ... // Second-order address (a^2)
... **a ... // Third-order  address (a^3)
...
```

An example of the use of Δ-annotation is given in figure 6.2. Any edge that operates on the statespace can be annotated with the Δ function, i.e. primitive and complex edges

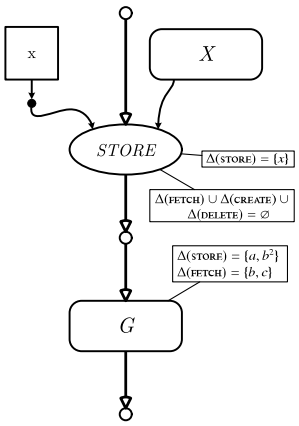


Figure 6.2: Annotations with Δ dependencies

alike. In the example, a primitive STORE edge is shown, that stores on some address x the value that graph X evaluates to. The annotations state that in the edge address x is used in a STORE operation and that the total set of addresses used in the FETCH, CREATE and DELETE operations is empty. The complex edge G performs *at least one* STORE operation on the addresses a and b^2 and FETCHS from addresses[1] b and c. No assertions are made about Δ(CREATE) and Δ(DELETE), which should not occur when compiling intelligently (see section 6.4), but which is completely valid, model wise.

6.2 Annotation-aware transformations

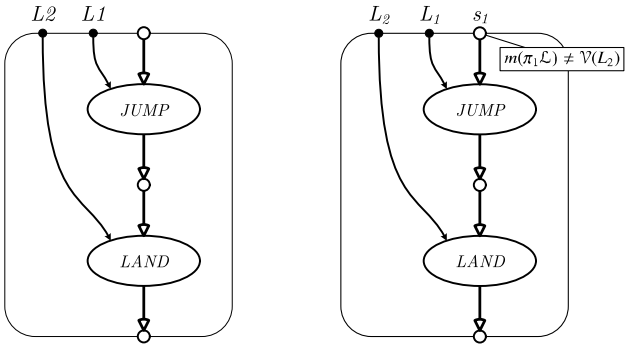
Recall the problem shown in figure 6.3a. The solution was overcome (section 4.4.5) by transforming the appropriate CREATE operation down to the local 'deadlock'. Unfortunately, this required transforming said CREATE across a splitting statespace (duplicating the operation in both branches) and the accompanying mux, thus enlarging the cumulative complexity of the transformations required. However, based on the exact same idea—that before a label's creation, no jump to said label is possible—annotations can be used with the same results, but without adding complexity.

Through propagation of assertions *through* the graph, as opposed to transformation *of* the graph, the transformation of the deadlocked locality can be made possible. Figure 6.3b shows the same situation, but with the added assertion that the active label is certainly not the same as the argument of the *Continue* operation. In this case the *Continue* can be dropped by the transformation, because whatever the state at s_1, the *Continue* will be given as an argument a state that has an active label, other than the other input of the operation.

Assertion propagation is less complex, because it is not required to propagate over all branches of the statespace. As observed above, directly after a CREATE operation, it is legitimate to assert that the active label can not be the label just created.

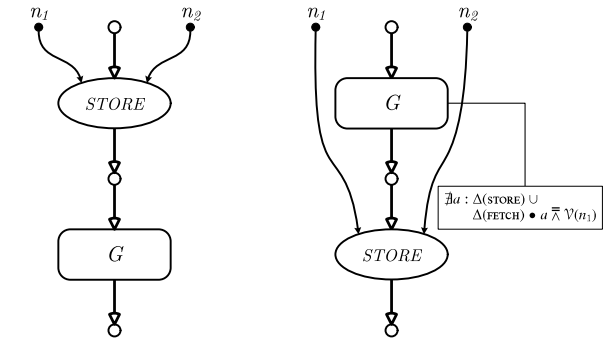
The true power annotations add to transformations is twofold; it lies in the added capability of evaluating across infinite recursion, on the one hand, and the weaker

[1] Although not guaranteed, in real world compilations—before any expansions—of c++, when any operation is performed on x^n there will always be a FETCH on x^{n-1}. Note that this is an inductive closure, so there will be FETCHS on all $b^1, b^2, \ldots, b^{n-1}$



(a) Repeated from fig. 4.5 (b) Added assertion

Figure 6.3: Annotation resolving transformation problem



(a) Potentially untransformable situation (b) Transformation after annotation

Figure 6.4: Annotation enabling transformation

transformation-definition requirement on the other.

Consider the situation shown in figure 6.4a. The depicted edge *G* is infinitely recursive, but assume for the example that *G* has been annotated with a complete definition of its Δ function. If the set of annotations of *G* can now be elaborated with the assertion that

$$\nexists a : \Delta(\text{STORE}) \cup \Delta(\text{FETCH}) \cup \Delta(\text{DELETE}) \bullet a \barwedge \mathcal{V}(n_1)$$

then it is safe to conclude that the STORE may safely be transformed over edge *G*. The resulting graph segment is shown in figure 6.4b.

The example shown in figure 6.4 also illustrates the notion of generalized transformation patterns. Since *G* can be any valid edge, it may just as well be a primitive edge. As long as the assertion used in the example can elaborate the annotations of *G* the transformation is valid, because the assertion is in itself sufficient for the proof of behavioral consistency of this graph before and after transformation. This means transformations can be expressed in this generalized form, because constraints on the transformations can be expressed in terms of assertions.

6.3 Proof of correctness

The proof of the correctness of transformations with Δ-annotations follows by inductive expansion of complex edges. Suppose for edge A and operation o that

$$\Delta(A, o) = X$$

for any given o. For any subdivision of A in edges B and C, it must—per definition—hold that

$$\Delta(B, o) \cup \Delta(C, o) = X$$

and thus

$$\Delta(B, o) \subseteq X \wedge \Delta(C, o) \subseteq X$$

. Transformations requiring Δ-annotations on a complex edge can likewise be split in multiple transformations on subgraphs B and C, since for any transformation requirement that

$$x \notin \Delta(A, o)$$

it can be propagated that

$$x \notin \Delta(B, o) \cup \Delta(C, o)$$

It is important to note that this assumes properly formulated annotations in terms of the in- and outputs. Thus if the assertion $a > b$ about edge A having input a and output b holds, it may very well not be the case that this literal constraint is valid for every edge in the defining graph class of A. The correctness, however, follows from observing that said assertion indicates a relation between whatever *nodes* correspond to a and b. If the assertion is sufficient for a transformation, it may thus be performed without having to expand A. Naturally, transformations still require individual proof.

Having said this, the general proof for all annotations is as follows: for consistent annotation-set N on edge A it is guaranteed that for any subdivision in subgraphs B and C of the expansion of A, both B and C may only be annotated by annotation m if it can legally elaborate (see 6.1) on N.

6.4 Compiling with annotations

During compilation many useful annotations can already be inserted on nodes and edges. The process of compilation with annotations can be compared to the conventional use of attribute grammars in general and S-attributed grammars in particular [3, chapter 3]. When a template is instantiated, all operations in the template itself are known, so their dependencies and the relations they define can be included in the annotation of the edge that represents an instance of the class created by this template. This statement implies a recursion, since a template mighty very well included placeholder edges that are filled in by further template instantiation. When these complex edges are filled in, they will have received annotations and from these the complete set of annotations can be composed for the template at the current level.

This does, of course, require some form of cycle detection. This is only required for the gathering of annotations from nested edges to top-level ones, since the instantiation

of templates is based purely on the input to the compiler, so for a finite input there is a finite instantiation cycle. Although complex, this is a field in which a lot of work has already been done and for the greater part of possible SILC-implementing compilers, known algorithms should suffice.

There is a design choose left to be made with respect to the annotation of infinitely recursive edges with nested creation *and deletion* of variables. It depends very much on the requirements of the final compiler how to treat this issue. It can well be argued that since a variable is created and deleted within the edge, it enforces no dependencies on the exterior of that edge. If so, these variables do not require inclusion in the annotation. On the other hand, when compilers are subject to cost-analysis and need to search for optimizations, throwing this information overboard is a bit reckless. Should the information be required a denotational method should be used to indicate the number of these 'hidden' variables in terms of eventual execution depth. Like the notation of higher-order addresses above, it is possible to use indexes to annotate these edges. As an example

$$\Delta(A, \text{CREATE}) = \{a, b, c\} \cup \{d_1, \ldots, d_n\} \cup \{e, f\}$$

indicates that the edge will CREATE a new d for everyone of the n recursions, but only a single instance of a, b, c, e and f.

The complete Δ-function can be generated from an edge by simply gathering all dependencies on a per-operator basis. For a given graph class G with edges n_1 through n_k the simple rule that

$$\forall\, o \left(\Delta(G, o) = \bigcup_{i=1}^{k} \Delta(n_i, o) \right)$$

The Δ is defined on graph class G here, which is not very formal, but it means to say that this definition is the definition for the Δ function on any *edge g* that instantiates graph class G.

Summary

Annotations add to the transformation capability of a graph by allowing transformations over infinitely recursive graphs. Their correctness follows from the *elaboration rule*. Besides the added power they give with respect to infinitely recursive graphs, they also allow for more lightweight transformations—by removing the necessity of expansion—over complex edges in general.

Transformation rules can be expressed in a more generalized way by using annotations in their formulation and proof.

7 Conclusions and recommendations

This chapter summerizes the most important conclusions from this thesis (7.1) and gives some recommendations for future work (7.2). For easy reference, all points have been enumerated.

7.1 Conclusions

1. SILC is capable of modeling pointers in such a way that the transformations are provably correct. Also, constant propagation—even for partially known (offset) pointers—is possible. The latter statement is more relevant to the implementation, because mathematically the solution new solution is equivalent to the old.
2. With the new model of storage in the statespace casts of pointers are supported. Casts of data in the graph itself has unchanged support (i.e. is supported).
3. SILC models all legal jumps in c++ in a scope-safe way.
4. With the given extensions SILC is still applicable to transformational design, since it still depends on provable transformations of a hypergraph.
5. By using annotations transformations become possible that would not be otherwise. These transformations mainly involve edges with infinite recursion.
6. Intuitively, annotations make many transformations a lot cheaper in terms of required computation (see 7.2.1/2).

7.2 Recommendations

7.2.1 Immediate followup

1. Obviously the remaining unmodeled language features should be looked into, to eventually arrive at a model that completely supports c++. Specifically:

(a) Model the function call mechanism especially with respect to jumps, as some function calls should still be performed when jumping over them, e.g. destructor calls when jumping out of scope.

(b) Find out whether there is a fundamental difference between the hidden execution of constructors and that of destructors and—if not—why the c++ standard does not allow for the first [5, 6.7/3] and *does* allow for the second [5, 6.6/2].

(c) Determine how much information is required at run-time for object-orientation and whether the amount for a formal description thereof differs.

2. Gather empirical data to determine what transformations form bottlenecks (either computationally or with respect the designer's intuition) and see whether other abstractions (using the same primitives—5.2) resolve these bottlenecks, also to investigate the bold statement about annotations in the conclusions (7.1/6).

7.2.2 Down the line

1. Since the hypergraph is itself a functional program, it deserves consideration to design mappings to and from the most common functional programming languages in the field. Also, to investigate whether SILC's extensions cover other imperative/object-oriented languages. This recommendation is important, because it seems possible to employ SILC as a *pan-paradigm translator*.

2. Some work has already been done with respect to the design of both a graph description language and a way to express the transformations. The most important conclusion drawn from the work so far is that a lot more work is required.

7.2.3 Implementation

1. A new tool should be developed that at least includes the extensions of SILC, but that would preferably allow for the addition of primitives and transformations that may be formulated later.

2. Even proof is a matter of trust. Therefor, it would greatly improve the quality of the development environment to provide a tool that checks the proofs of transformations (or that calculates the proof itself). The implementation of such a tool is, of course, heavily dependant on the language used (7.2.2/2).

A

High-level Synthesis based on Transformational Design

Thijs Krol and Bert-Steffen Visser

Abstract. Due to the ever-increasing complexity of digital systems, the verification of the various steps in the design flow is becoming more and more a burden. Lengthy simulation runs do never guarantee the correctness of a design. Moreover, formal verification techniques are not yet generally applicable. This paper describes a part of a design flow for high-throughput digital signal processing applications from an executable specification in terms of a Kahn model of processes described in C to a signal flow graph description from which the scheduling and resource allocation can start. This design flow is based on an alternative way for obtaining correctness-by-construction. In this so-called 'Transformational design method', correctness is obtained by applying small local behavior-preserving transformations, which have themselves already been proven correct, to derive an implementation from an executable specification. Essential to the transformational design method is the design representation and its formal semantics for proving the behavior-preserving property of the transformations. The small local behavior-preserving transformations and the possibility to switch between different time models and different levels in the functional and data hierarchy makes it easy to explore many design alternatives without losing the correctness property. This will improve the quality of the design, the design time and the controllability of the design process.

A.1 Introduction

The design of large digital (sub)systems is characterized by a specification phase and a design phase. The latter can be divided in a high-level synthesis part and a low-level synthesis part. In the design of high-throughput digital signal processing applications

for video processing, mobile telephone, robot controllers, etc. hardware/software co-design is governing [2]. In this application domain, during the specification phase, in most cases a simulation model of the system is constructed allowing the verification by means of simulation of the specification and algorithms against the idea, [6,8]. In practice, for performance reasons, these specifications are often written in C based on some model of communicating modules [6]. Starting from such an executable specification, the functionality expressed by the specification and its underlying model is mapped on a target architecture. This target architecture might either be chosen beforehand or may be the result of the high-level design process. During the high-level synthesis process, tasks such as decomposition, refinement, dependency analysis, scheduling, and resource allocation are performed. The result is a design description, in VHDL, Verilog or some proprietary language that is rather close to the final design, though it still needs to be optimized. The low-level synthesis process finalizes the design by re-timing, logic optimization, placement and routing, etc.

Even in this domain of high-throughput digital signal processing applications, a large variety of high-level design flows is used, varying from manual design for which many tools are available to full silicon-compilation. All these design flows suffer from design faults, either induced by 'the creativity of' the designer or by the complexity and immaturity of the design tools. Hence, all steps in the design flow have to be verified by means of lengthy simulations in which the behavior before and after the design step is compared. Only in few cases formal verification can be applied. The more complex the systems to be designed are, the more severe the verification problem is.

This paper discusses research on an alternative design flow that is based on correctness-by-construction and which relieves the simulation burden. In this so called 'transformational design method', the design flow consists of the application of a large number of simple pre-proven behavior-preserving transformations. The feasibility of such a design flow has been shown [13, 14]. However, due to the immaturity and incompleteness thus far, of the required design tools, it has never been applied to real large designs.

High-level synthesis tools for high-throughput applications, such as the Phideo tools [9], in general start from some kind of signal flow graph (SFG). For example, the Phideo tools start from a SFG that expresses a number of execution units and the multi-rate data flow between them and perform the high-level synthesis tasks such as scheduling, resource allocation and the generation of the control unit. Till now, the translation of the executable specification in terms of communicating processes described in C, to the SFG and the derivation of the execution units is done manually. The efficiency of the final design strongly depends on the design decisions taken during this translation. Clearly this manual part of the design flow is extremely error prone.

This paper focuses on a transformational design flow for high-throughput digital signal processing applications, starting from an executable specification in terms of a Kahn model [5] of communicating processes, described in C, and resulting in a synchronous control data flow graph (CDFG). From this point on existing tools are available to complete the high-level synthesis process. The transformational design method provides correctness-by-construction and gives full freedom to the designers' creativity.

The idea of correctness-by-construction is not new at all. In fact any automated design flow from executable specification to a description in terms of gates and registers could be correct if all tools were correct and only correct manual interference was possible. In practice, however, this is difficult to realize. Some tools cannot be trusted

because they are immature or too complex. The design representations of different tools often do not match and often the semantics of tools and representations are ambiguous. Hence, lengthy simulations are needed to verify the various design steps.

Based on these observations an alternative approach is investigated [3, 13, 14] in which the design flow is divided into small preproven behavior preserving transformations. These transformations can be initiated by the designer or be the result of some optimization algorithm. The method could be compared with a theorem prover, the designer decides on the algebraic and logical rules to be applied and the theorem prover provides suggestions and checks for illegal steps.

Such a transformational design method must be based on a formal model in which structure, behavior (including control) and time can be expressed. This model should support a 'language' for describing specification, design and transformations. In our case, the model and the language should at least encompass both the specification level in terms of a Kahn model with processes described in C and the implementation level in terms of gates, adders, registers etc. Furthermore, the model must be compositional, i.e. when a part of the design description is replaced by a different part with the same external behavior, the external behavior of the total design remains unchanged.

In section 2 we will describe the language that supports the design flow, its expressiveness and the informal semantics. In section 3 we will deal with the two paradigms for modeling time, i.e. the (clock)synchronous that relates to an implementation in registers and timeless functions such as gates and adders, and the asynchronous paradigm, in which Kahn model and the C processes are described. In section 4 the semantical model is described.

A.2 The Language

A.2.1 Hyper-graphs

Because both structure and behavior need to be expressed we have chosen for design representation in the form of a Control Data Flow Graph, in which control and data are modeled in the same way [7]. Usually, the nodes in a control data flow graph stand for operations and the connections between the operations are represented by edges. Because the operations have ports (inputs and outputs) that should be distinguished, the edges need to be annotated with the port identifiers of the operations that are modeled by the nodes. In case of multiple undirected connections, without special precautions this results in non-unique representations. Therefore we have chosen for hyper graph model [4], which elegantly describes a signal flow graph and better fits to our semantical model.

A hyper graph consist of nodes which are connected by means of hyper edges that have at least one extremity, see figure A.1-A, B and C. These extremities are annotated with identifiers and thus can be distinguished. So a hyper edge is just an edge that connects one, two or more nodes. Notice that in a normal graph the edges always have two extremities identified by 'begin' and 'end'.

The hyper edges in the graph symbolize operations and the nodes represent the operands. We attribute a structural interpretation to the hyper graph. So the hyper graph describes the way in which operators are connected. The nodes are to be considered as connection points. Therefore, we assume that the data values on which the hyper edges operate, are 'bound' to the nodes (connection points). A value bound to a node x is denoted by $\mathcal{V}(x)$. So x is a structural identifier, not to be mistaken with

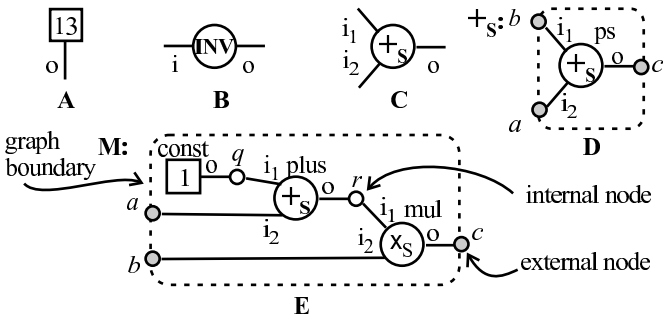


Figure A.1: A hyper-graph and its components.

the dummy variable representing a data value used in algebraic expressions. The hyper edges (operations) are colored with an identifier, written inside the edge, which refers to the operation it performs and are identified by a name written outside the node.

Hyper edges are represented in an intuitive way as is given in the figures A.1-A, B and C. In figure A.1-E a complete graph M is represented that performs the function $\mathcal{V}(c) = (\mathcal{V}(a) + 1).\mathcal{V}(b)$. The graph is built from three hyper edges. Each hyper edge puts a restriction on the values bound to the nodes to which the hyper edge is connected. The first hyper edge, the square, with one extremity says $\mathcal{V}(q) = 1$. The hyper edge annotated with '$+_S$' states $\mathcal{V}(r) = \mathcal{V}(a) + \mathcal{V}(q)$ and the last one states $\mathcal{V}(c) = \mathcal{V}(b).\mathcal{V}(r)$. The nodes q and r are internal nodes that cannot be influenced from the outside. The nodes a, b and c are external nodes and define the external behavior of the graph, i.e. the relation between the values bound to the external nodes, viz. $\mathcal{V}(c) = (\mathcal{V}(a) + 1).\mathcal{V}(b)$. The internal behavior (or just behavior) of the graph describes the relation between the values bound to all nodes of the graph and is thus given by the relations $\mathcal{V}(q) = 1$, $\mathcal{V}(r) = \mathcal{V}(a) + \mathcal{V}(q)$ and $\mathcal{V}(c) = \mathcal{V}(b).\mathcal{V}(r)$. It is important to notice that these are relations that do not induce an evaluation order. So, in first instance the hyper graph makes no distinction between inputs and outputs. However, in practice it is useful to distinguish between input and outputs by annotating the extremities of the hyper edges with an arrow point, see for example figure A.2.

A graph represents external behavior in the form of a set of relations on the values bound to the external nodes of the graph. The external behavior follows from the behavior of the graph, which is given in the form of a set of relations on the values bound to all nodes of the graph. These relations follow from the hyper edges. The behavior of a hyper edge is again defined by the external behavior of its defining graph.

Graphs are either primitive or non-primitive. A primitive graph only consist of one hyper edge of which the behavior is defined by the graph itself, figure A.1-D. Clearly, the behavior of a primitive graph must be defined explicitly, for instance by a predicate or a piece of program in some suitable programming language. For example, the primitive graph representing a saturating addition on the unsigned integers $\{0 \cdots 2^{16} - 1\}$ is defined by figure A.1-D and the relation:

$$\begin{aligned} &\mathcal{V}(a), \mathcal{V}(b), \mathcal{V}(c) \in \{0, \cdots, 2^{16} - 1\} \\ &\mathcal{V}(c) = \begin{cases} \mathcal{V}(a) + \mathcal{V}(b) & \text{if } \mathcal{V}(a) + \mathcal{V}(b) \leq 2^{16} - 1 \\ 2^{16} - 1 & \text{if } \mathcal{V}(a) + \mathcal{V}(b) > 2^{16} - 1 \end{cases} . \end{aligned} \tag{A.1}$$

The language has a graphical representation instead of a lexical representation. A de-

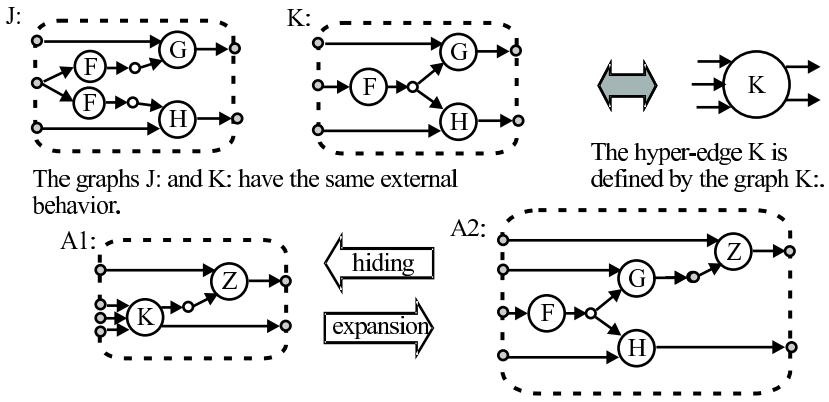


Figure A.2: Graphs with identical external behavior. Expansion and hiding

sign is 'written' by means of a graphical user interface or is the result of a compilation of an other representation such as C or VHDL. The design representation is stored in a database. The 'syntax rules' of a lexical representation are in this case a set of rules that should be satisfied by the database content. Examples of such rules are: 'In a graph an extremity of a hyper edge must always be connected to a node'. A formal model in terms of sets and relations supports the database content.

A.2.2 Functional Hierarchy, Recursion and Repetition

From the preceding sections we learned that the basic operations, comparable with the basic arithmetic operations in computer languages are represented by primitive graphs. Any graph represents external behavior and so does the behavior of a hyper edge. With these hyper edges new graphs can be constructed, again defining new hyper edges. So a hyper edge is an instantiation of a graph. This implements both structural and functional hierarchy.

Different graphs can have the same external behavior, such as the graphs J: and K: in figure A.2. So it makes no difference which graph is used for defining the behavior of the hyper edge that is defined by one of these graphs. The external behavior of the graph in which the hyper edge is instantiated remains the same. This is the basic principle for design transformations, i.e. changing the structure while preserving the external behavior.

Clearly, a hyper edge may be replaced by its defining graph (expansion) and conversely a part of a graph may be replaced by a hyper edge (hiding). In the latter case, a graph that defines the hyper edge should be added to the design. Figure A.2 shows the expansion and hiding process. Notice that expansion and hiding are design transformations too.

If a graph calls itself, directly or indirectly, this implements recursion. Recursion is the only way to implement repetition in our transformational design language. Notice that we are dealing with structural recursion, i.e. recursion is to be interpreted by expanding the graph. Consequently, the expansion will result in a semi-infinite array of hyper edges and nodes, see figure A.3.

In practice all repetitions are finite, otherwise we started from an incorrect specification. So only a finite number of the instantiated hyper edges in the expanded graph

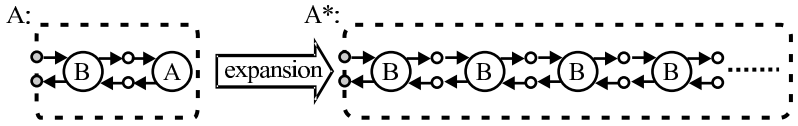


Figure A.3: Expansion of a recursively defined graph.

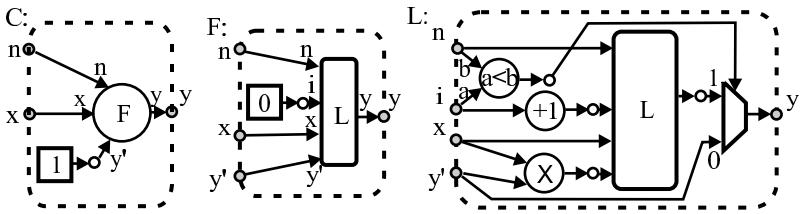


Figure A.4: The representation of [`y=1; for(i=0;i<n;i++) y=y*x;`]

will contribute to the external behavior of the graph. The remaining edges can be removed by means of dead code elimination. This will be illustrated in the next example in which we will model a while loop.
Consider the following piece of C-code that implements $y = x^n$.

```
y=1; for(i=0;i<n;i++) y=y*x;
```

Figure A.4 shows the translation of this piece C-code to our design representation. The code fragment has two input variables x and n and one output variable y and is symbolized by the graph C in which the hyper edge F stands for the for-loop. The graph F consists of an initializing part $i = 0$ and a repetition part given by the hyper edge L. The graph L implements the loop in a recursive way. It checks whether $n < i$ and if so then the result of the next iterations is sent to the output y, otherwise the current value of y' is sent to the output y. Notice that the translation from C to the control data flow graph representation results in single-assignment-code. Recall that we are dealing with structural recursion, so the behavior of the graph follows from expanding the graph. When the expansion transformation is applied twice on the graph 'L:' in figure A.4, we obtain the graph shown in figure A.5. If n is a constant, for example $n = 2$, we can reduce the graph by means of the transformations 'constant propagation' and 'dead code elimination'. The result of the constant propagation is shown by the values in the gray squares. After dead code elimination only the gray hyper edges remains, resulting in the graph at the bottom of the figure.

A.2.3 Data hierarchy and data types

Besides the functional hierarchy described in the preceding section, we also need data hierarchy in order to be able to model complex data structures. In ordinary programming languages for this purpose arrays and structures are used. In our design language we have one generic construct that allows us to model any form of data hierarchy. This construct is based on the observation that arrays and structures can be mathematically modeled by functions.

In our language we distinguish between two classes of data values: scalar types and

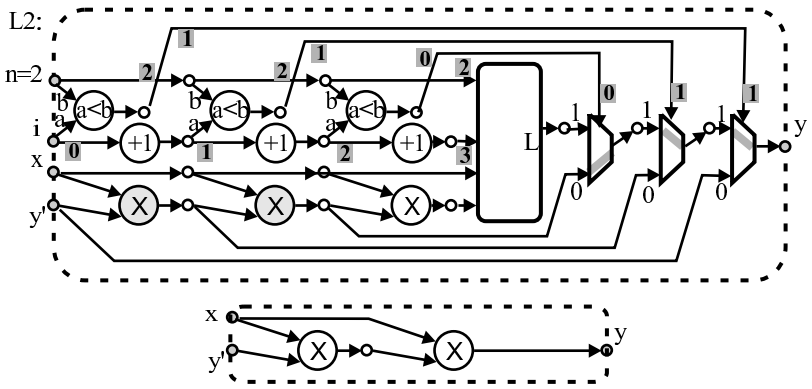


Figure A.5: The two times expanded version of L (in figure A.4)

structured types. Scalar types are Booleans $\mathcal{B} = \{\mathbf{0}, \mathbf{1}\}$, integers $\mathcal{Z}$, bounded integers $\mathcal{Z}_{a,b} = \{x \mid a \leq x \leq b\}$ and identifiers. The Booleans, bounded integers and identifiers are finite sets. The term scalars refers to set *Scalars* that is the union of the sets of Booleans, integers, bounded integers and identifiers.
A function f is defined as a set of tuples $\langle a, b\rangle$, in which a is an element of *Scalars* and b is an element of *Scalars* or b is a function that can only be constructed according to this definition, and that satisfies

$$\left[\ \forall a, b, c : (\langle a, b\rangle \in f \wedge \langle a, c\rangle \in f) \Longrightarrow b = c\ \right]. \tag{A.2}$$

Examples of functions are $\{\langle 0, \mathbf{0}\rangle, \langle 1, 3\rangle\}$ and $\{\langle 0, 0\rangle, \langle 1, \{\langle 0, \mathbf{0}\rangle, \langle 1, 3\rangle\}\rangle\}$.
The class of structures comprises all functions constructed according to the previous definition.

In algebraic expressions the variables are dummy variables. Dummy variables are usually denoted by an italic font and values by a normal font. So in $x = a + 10$, x and a are dummy variables written in italic and 10 is a value written in a normal font. In the algebraic expressions in this paper the same convention is used. So if a value or identifier refers to a particular object, it is written in normal font.

The domain dom(f) and the image im(f) of a function f are defined by:

$$\text{dom}(f) = \{a \mid \exists b : \langle a, b\rangle \in f\} \ \text{ and } \ \text{im}(f) = \{b \mid \exists a : \langle a, b\rangle \in f\}. \tag{A.3}$$

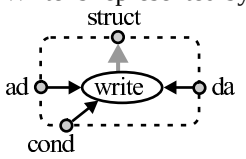
So dom(f) is always a subset of *Scalars* and the image im(f) is a subset of the union of the scalars and the structures.

With this definition of functions, an array is modeled as a function on the bounded integers, for example an array of length 3 containing the values 0, 3 and 5 is modeled by $\{\langle 0, 0\rangle, \langle 1, 3\rangle, \langle 2, 5\rangle\}$ and a structure consisting of the family name and a list of the children's first names is modeled by:

$\{\langle$family-name, Smith$\rangle, \langle$children, $\{\langle 0,$ John$\rangle, \langle 1,$ Christine$\rangle, \langle 2,$ Basil$\rangle\}\rangle\}$

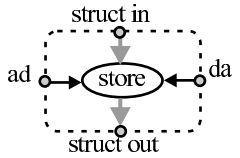
In order to build structures, four operations are available in our design language, viz.: Write, Store and Fetch.
The Write and Fetch are primitive. The Store is non-primitive but for simplicity we will treat it as a primitive graph. These graphs are defined as follows:
The Write states that the set containing the tuple $\langle \mathcal{V}(\text{ad}), \mathcal{V}(\text{da})\rangle$ is a subset of the

value bound to 'struct', provided its condition input is **1**, in which $\mathcal{V}$(ad) is the address value bound 'ad' and $\mathcal{V}$(da) is the data value bound 'da'. If the value bound to 'cond' is **0**, it states that the empty set is a subset of value bound to 'struct'. The latter of course is always true. Notice that two hyper edges Write with the same address values and different data values and their 'struct' extremities connected to the same node, is incorrect and causes the entire design description to be incorrect.
The Write is represented by:



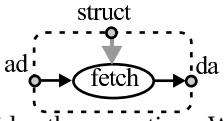
$\mathcal{V}(\text{cond}) = \mathbf{1} \Longrightarrow \{\langle \mathcal{V}(\text{ad}), \mathcal{V}(\text{da})\rangle\} \subset \mathcal{V}(\text{struc})$
$\mathcal{V}(\text{cond}) = \mathbf{0} \Longrightarrow \varnothing \subset \mathcal{V}(\text{struct})$
$\mathcal{V}(\text{struct})$ is a function

The Store adds on 'struct out' a tuple $\langle \mathcal{V}(\text{ad}), \mathcal{V}(\text{da})\rangle$, to the value bound to 'struct in'. If the input 'struct-in' already contains a tuple $\langle \mathcal{V}(\text{ad}), x\rangle$, for any x, this tuple is first removed. So the Store reflects the over-writing semantics of the store operation in an ordinary sequential computer language.
The Store is represented by:



$\mathcal{V}(\text{struct out}) =$
$(\mathcal{V}(\text{struct out}) \upharpoonleft \{\mathcal{V}(\text{ad})\}) \cup \{\langle \mathcal{V}(\text{ad}), \mathcal{V}(\text{da})\rangle\}$
In which $F \upharpoonleft A = \{\langle a, b\rangle \mid \langle a, b\rangle \in F \wedge a \notin A\}$

The Fetch reads the value bound to the address given by the value, bound to the address input 'ad', and delivers this value at the output 'da'. The Fetch is identical to the Write with condition input **1**. The only difference is the order of evaluation expressed by the arrow points.
The Fetch is elucidated below.



$\{\langle \mathcal{V}(\text{ad}), \mathcal{V}(\text{da})\rangle\} \subset \mathcal{V}(\text{struc})$

Besides the operations Write, Store and Fetch some other operations are defined for example for modeling the scoping and the heap in C.

A.2.4 Translating C code to a Flow graph Model

We assume that all variables used in the C program are stored in a struct D. This struct D represents the data space of the process. The statements and expressions in C successively operate on this struct. So, if a variable a in C has the value 5 then $\langle a, 5\rangle \in$ D. Variables local to some C function have their own struct. Variables have to be initialized during declaration. So a declaration '`int a = 10;`' corresponds to a store of a constant 10 on the address 'a' in the struct D, figure A.6-A.
In a statement such as '`b = a+1`', only the variables a and b are affected. First the variable a has to be fetched from address 'a' from the struct D, then a constant 1 has to be added and finally the result has to be stored at address 'b', figure A.6-B.
The basic idea behind the translation is that each statement in C results in a unique graph. The hyper edges the graph follow from the statement or the substatements. For example the if-statement

`if (<expr>) <statement>`

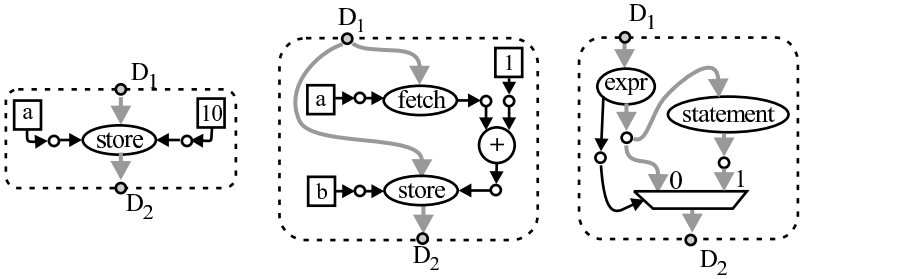


Figure A.6: Some translations from C to a control data flow graph

is translated into a graph consisting of a multiplexer, one hyper edge that models the expression, which might have side effects and thus might update the struct, and one that models the statement. If the expression evaluates to true the output-struct of the statement is chosen else the output-struct of the expression is chosen as the final result of the if-statement.

A.3 Modeling Time

A specification in the form of a Kahn model [5] consists of a number of communicating processes, each process is specified in sequential language and these processes communicate via channels. However, the final design can be fully (clock) synchronous, which is easier to describe. For this reason, our design language is based on two different paradigms, viz. the synchronous paradigm and the process paradigm. A transformation from the synchronous paradigm to the processes paradigm is always possible. The transformation from the processes paradigm to the synchronous paradigm is possible if the communicating processes satisfy the restrictions pointed out in section A.3.2.

A.3.1 The Synchronous Paradigm

The synchronous model is based on finite state machines at the register transfer level. Our design language describes the system in terms of functions and unit delays (registers). The functions are timeless and the delay operation @ transfers the data from to the next instance of time. The most general form is elucidated in figure A.7 together with its unfolding in time. The function F calculates each instance of time the output value b and the new state sn from the input a and the old state so. We assume time starts at $t = 0$, hence the old state at $t = 0$ has to be provided explicitly by means of an additional initialization input *init* at the delay operation @. So the model is functionally described by:

$$\begin{aligned} &\mathcal{V}(\mathrm{so})_0 = \mathit{init}\ ,\ \mathcal{V}(\mathrm{so})_t = \mathcal{V}(\mathrm{sn})_{t-1} \ \text{ for } t \in \mathcal{Z} \text{ and } t \geq 1 \\ &(\mathcal{V}(\mathrm{b})_t, \mathcal{V}(\mathrm{sn})_t) = F(\mathcal{V}(\mathrm{a})_t, \mathcal{V}(\mathrm{so})_t) \ \text{ for } t \in \mathcal{Z} \text{ and } t \geq 0 \end{aligned} \tag{A.4}$$

The values at the internal and external nodes are described in terms of the values at a time instance t, but in fact describe semi-infinite sequences $x_0, x_1, x_2, \ldots$.

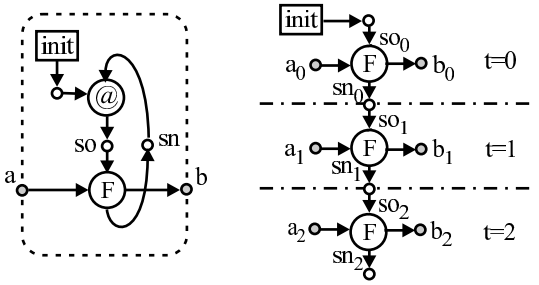


Figure A.7: The synchronous model and its unfolded version

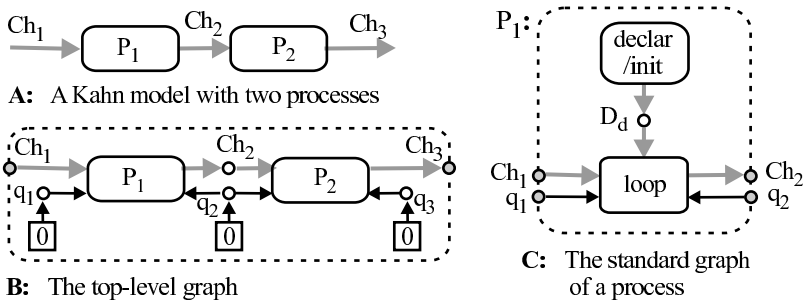


Figure A.8: The Kahn model translated into a control data flow graph.

A.3.2 The Process Paradigm

The process paradigm is inspired by the Kahn model in which a number of processes communicate with each other and with the environment via channels. These channels are FIFO buffers that connect the processes one to one. A process can always write to a channel (non-blocking write) and if a process tries to read a value that is not yet available it will wait (blocking read). We assume the Kahn model is deadlock free and no starvation or flooding can occur. The verification of this, which is far from trivial, is left to the system design process that precedes the specification. So a process delivers a sequence of values to a channel and another process reads the successive values. The relation between the sequences of values is determined by the processes.

In our design language, following the process paradigm, a process is modeled by a hyper edge and a node to which a sequence is bound models a channel. A sequence *Ch* is a semi-infinite array, i.e. a struct *Ch* with domain dom(*Ch*) = $\{i \mid i \in \mathcal{Z} \wedge i \geq 0\}$. Writing to, and reading from the channel (sequence) is done by means of the Write and Fetch operations, cf. section A.2.3.

The way in which the process paradigm is implemented in detail and the way in which a specification in the form of a Kahn model with processes described in C, is translated into our design language, is illustrated in the following example:

Consider a simple Kahn model with two processes P_1 and P_2, a channel that sends data from P_1 to P_2, an input channel and an output channel, as is shown in figure A.8-A. The processes are described in C. Each process is modeled by a hyper edge, see figure A.8-B. The translation from C to the graph defining these hyper edges follows

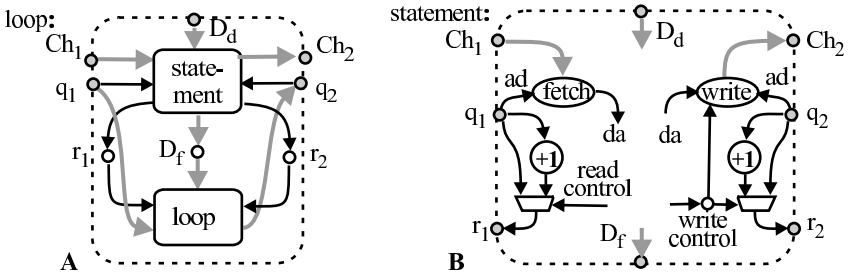


Figure A.9: The translation of the while loop and the read and write operations.

the method described in section A.2.4. The channels in the Kahn model are described in C++. The channels are instantiations of a class with methods <buffer_id>.write and <buffer_id>.read. Each instantiation of this class is translated to a sequence, i.e. a semi-infinite array, together with an integer that points to the 'data value in progress', see figure A.8-B. The integers are initialized to 0.

The processes in C, in figure A.8-A, are modeled by infinite loops, for instance 'while(1) <statement>'. Before a process runs into the loop, it is initialized by variable declaration and initialization. So the hyper edge P_1 is defined by a graph 'P_1 :', figure A.8-C, consisting of two hyper edges, one that models the declarations and initializations, 'declar/init', and one that models the loop. All variables are modeled by a struct bound to node D_d, cf. figure A.6-A. Clearly, all processes are equally structured at this level of hierarchy.

The hyper edge 'loop' is defined by a recursive graph, figure A.9-A, and consists of a hyper edge 'statement' that models the statements in the while-loop and a hyper edge 'loop', which is defined by the graph itself. The channels are directly connected to both the hyper edges 'statement' and 'loop', so these hyper edges can operate independently on the channels. The integer that is associated with a channel, for example the integer that is bound to the external node q_1, is connected to the hyper edge 'statement' where it is updated according to the number of data values of the sequence that have been read inside the hyper edge 'statement'. The resulting value of this integer is bound to r_1 in order to be used in the next instantiation of 'loop'. So integers associated with the channels indicate the index of the value to be read or written next. The number of reads and writes in each instantiation of 'statement' may be data dependent, as is depicted in figure A.9-B.

A.3.3 The Relation between the Synchronous Model and the Process Model

Generally, the transformational design flow starts with a specification in terms of communicating processes, the process model, and ends with a design description in terms of registers and functions, the synchronous model. So, somewhere in the transformational design flow we have to transform the process model to the synchronous model. For this purpose the design description in terms of the process model will be transformed such that all graphs 'statement' contain exactly one unconditional 'fetch' or 'write' connected to each channel, see figure A.9-B. Such a graph corresponds to the synchronous model depicted in figure A.7. The hyper edge 'declar/init' in figure A.8-C

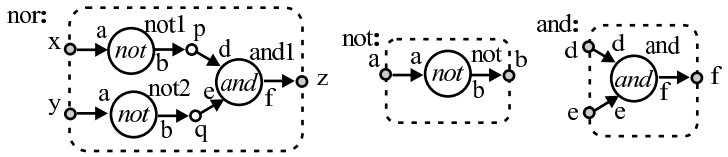


Figure A.10: A NOR-gate implemented from two OR-gates and an AND-gate.

corresponds to 'init' in figure A.7 and the hyper edge 'statement' in figure A.9-A corresponds to the hyper edge 'F' in figure A.7. The equivalence of both models follows immediately from unfolding the recursion in figure A.9-A and the unfolded interpretation of the synchronous model shown in figure A.7.

A.4 The Semantical Model

The design flow we propose, is based on correctness-by-construction, meaning that the external behavior expressed by the final design satisfies the external behavior expressed by the specification. The entire flow from specification to final design is performed by small behavior-preserving transformations. Clearly, a mathematical model should support such a transformational design system, to guarantee the correctness of the transformations and to obtain an unambiguous interpretation of our design language. The mathematical model that is used was originally developed for the description of relational databases [1]. The definitions used in that theory have been generalized and extended in order to cope with the structures described in section A.2.3. In the scope of this paper, it is impossible to elaborate thoroughly on the semantical model, so we will only give an overview of the approach.
Just like the language, the mathematical model should support both structure and behavior.

A.4.1 Structure.

A design *Des* consists of a number of graphs G, $G \in Des$. Each graph G is built from a set of input nodes, $N_{inp,G}$, a set of output nodes, $N_{out,G}$, a set of internal nodes, $N_{int,G}$ and a set of hyper edges E_G. The hyper edges are related to their defining graphs by a function g_G, with $g_G \in (E_G \rightarrow G)$. So $g_G(e)$ is the graph that defines e. The external nodes of its defining graph name the extremities of a hyper edge. The way in which the extremities are connected to the nodes of the graph is described by a bijective function $h(e)$. This function, called the instantiation function, maps the nodes of the graph connected to the hyper edges onto the external nodes of the defining graph of the hyper edge. The example in figure A.10 illustrates this. It is defined by the sets:

$$
\begin{array}{lll}
\text{Des} = \{\text{nor :, not :, and :}\} & & \\
N_{inp,\text{nor:}} = \{\text{x, y}\} & N_{out,\text{nor:}} = \{\text{z}\} & N_{int,\text{nor:}} = \{\text{p, q}\} \\
N_{inp,\text{not:}} = \{\text{a}\} & N_{out,\text{not:}} = \{\text{b}\} & N_{int,\text{not:}} = \varnothing \\
N_{inp,\text{and:}} = \{\text{d, e}\} & N_{out,\text{and:}} = \{\text{f}\} & N_{int,\text{and:}} = \varnothing \\
E_{\text{nor}} = \{\text{not1, not2, and1}\} & E_{\text{not}} = \{\text{not}\} & E_{\text{and}} = \{\text{and}\}
\end{array}
\tag{A.5}
$$

the functions g_G that provide the defining graphs of the edges:

$$\begin{array}{lll} g_G(\text{not1}) = \text{not}: & g_G(\text{not2}) = \text{not}: & g_G(\text{and1}) = \text{and}: \\ g_G(\text{not}) = \text{not}: & g_G(\text{and}) = \text{and}: & \end{array} \tag{A.6}$$

and the instantiation functions:

$$\begin{array}{ll} h(\text{not1}) = \{\langle \text{x, a}\rangle, \langle \text{p, b}\rangle, \} & h(\text{not}) = \{\langle \text{a, a}\rangle, \langle \text{b, b}\rangle\} \\ h(\text{not2}) = \{\langle \text{y, a}\rangle, \langle \text{q, b}\rangle, \} & h(\text{and}) = \{\langle \text{d, d}\rangle, \langle \text{e, e}\rangle \langle \text{f, f}\rangle\} \\ h(\text{and1}) = \{\langle \text{p, d}\rangle, \langle \text{q, e}\rangle, \langle \text{z, f}\rangle\} & \end{array} \tag{A.7}$$

These set and relations fully describe the structure of the design. For simplicity, all identifiers are named globally.

A.4.2 Behavior

The values are bound to the nodes in the graph, i.e. to the structural identifiers. A set of values bound to the nodes satisfying the behavior of the graph is called an observation. Hence, an observation is a function from the structural identifiers, the nodes of the graph, to the data values. These data values can have different types. So, one observation of the graph 'not:' in figure A.10 is $\{\langle \text{a, } \mathbf{0}\rangle, \langle \text{b, } \mathbf{1}\rangle\}$. The behavior of a graph is defined as the set of all possible observations of the graph and is denoted by $Beh(G)$. So, the behavior of the graph 'not:' is given by

$$Beh(\text{not}:) = \{\{\langle \text{a, } \mathbf{0}\rangle, \langle \text{b, } \mathbf{1}\rangle\}, \{\langle \text{a, } \mathbf{1}\rangle, \langle \text{b, } \mathbf{0}\rangle\}\} \tag{A.8}$$

Such a set of functions all having the same domain is called a Table. This name follows naturally from the representation of such a set of functions.

$Beh(\text{not}:) =$

a	b
0	**1**
1	**0**

The domain Dom(T) of a table T is by definition the domain of the functions it contains. In the preceding example Dom(Beh(not :)) = {a, b}.

Instantiation of behavior:
In the graph 'nor:', the observations on the nodes x and p have to agree with the observations of the graph 'not:' because the graph 'not:' prescribes the behavior of the hyper edge 'not1'. The observations on the nodes x and p therefore must be an element in Beh(not :) where a is substituted by x and b is substituted by p. Consequently, the elements in the domain of the table are renamed.
The domain of a function f can be renamed by the function composition $f \circ h$ in which h is a bijection mapping the new names on the old ones. Function composition is defined by:

$$f \circ h = \{\langle a, c\rangle \mid \exists b : \langle a, b\rangle \in h \wedge \langle b, c\rangle \in f\} \tag{A.9}$$

The domain of a table is renamed similarly by table-function composition $T \infty h$, in which the function h is a bijection. Likewise, table-function composition is function composition of all functions in the table, i.e.:

$$T \infty h = \{t \mid \exists f : f \in T \wedge t = f \circ h\} \tag{A.10}$$

So, the restriction put by hyper edge 'not1' on the observations on the nodes 'x' and

'p' is given by the table *Beh*(not1) which follows from:

$$\begin{aligned} \mathit{Beh}(\text{not1}) &= \mathit{Beh}(\text{not :}) \infty \{\langle \text{x, a}\rangle, \langle \text{p, b}\rangle\} \\ &= \begin{array}{|c|c|}\hline \text{a} & \text{b} \\ \hline \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} \\ \hline \end{array} \infty \{\langle \text{x, a}\rangle, \langle \text{p, b}\rangle\} \\ &= \begin{array}{|c|c|}\hline \text{x} & \text{p} \\ \hline \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} \\ \hline \end{array} \end{aligned} \tag{A.11}$$

The behavior *Beh*(not1) of the hyper edge 'not1' is called the instantiation of the behavior *Beh*(not :) of its defining graph 'not:'. The function that renames the domain of the table in equation (A.11) is called the instantiation function. The function *h*(not1) given in (A.7) describes the mapping from the nodes to which the hyper edge is connected to the external nodes of its defining graph. So, the behavior of a hyper edge *H* that is defined by a graph *G* is determined by:

$$\mathit{Beh}(H) = \mathit{Beh}(G) \infty h(H) \tag{A.12}$$

Deriving the behavior of a graph from its hyper edges:
Each of the hyper edges puts a restriction on the values that can be observed on the nodes in the graph. Or stated differently, an observation on the graph satisfies the behavior of all its hyper edges. For example, the observation $\begin{array}{|c|c|c|c|c|}\hline \text{x} & \text{p} & \text{y} & \text{q} & \text{z} \\ \hline \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \hline \end{array}$ of the graph 'nor:' satisfies the behavior of the hyper edges, because the observation $\begin{array}{|c|c|}\hline \text{x} & \text{p} \\ \hline \mathbf{0} & \mathbf{1} \\ \hline \end{array}$, $\begin{array}{|c|c|}\hline \text{y} & \text{q} \\ \hline \mathbf{1} & \mathbf{0} \\ \hline \end{array}$ and $\begin{array}{|c|c|c|}\hline \text{p} & \text{q} & \text{z} \\ \hline \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \hline \end{array}$ are elements in the behaviors of the hyper edges 'not1', 'not2' and 'and1' respectively.
Notice that these four observations are functions. Any pair of these functions is called compatible because the function values of the elements in the intersection of their domains are the same. So, two functions f and g, in which $f(x)$ and $g(x)$ are scalars, are compatible if:

$$\forall x : x \in (\text{dom}(f) \cap \text{dom}(f)) \Longrightarrow f(x) = g(x) \tag{A.13}$$

In case the function values $f(x)$ and $g(x)$ are of the structured type, the definition of compatibility has to be extended. The same holds for the definition of the natural join $\bowtie$ below. These definitions are rather complex and beyond the scope of this paper.
The union of two functions, generally is not a function. However, the union of two compatible functions is a function.
From the preceding we observe that an observation in the behavior of a graph must be compatible with at least one observation in the behavior of each of the hyper edges in that graph. And thus, the behavior of a graph follows from all compatible observations of the hyper edges.
This leads to the following definition of the natural-join of two tables T and U

$$T \bowtie U = \{t \cup u \mid t \in T \wedge u \in U \wedge t \text{ and } u \text{ are compatible }\} \tag{A.14}$$

For example the behavior of the graph 'nor' is calculated by:

$$\mathit{Beh}(\text{nor :}) = \mathit{Beh}(\text{not1}) \bowtie \mathit{Beh}(\text{not2}) \bowtie \mathit{Beh}(\text{and1}) \tag{A.15}$$

or

x	p
0	**1**
1	**0**

$\bowtie$

y	q
0	**1**
1	**0**

$\bowtie$

p	q	z
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

$=$

x	p	y	q	z
0	**1**	**0**	**1**	**1**
0	**1**	**1**	**0**	**0**
1	**0**	**0**	**1**	**0**
1	**0**	**1**	**0**	**0**

(A.16)

From the preceding, we conclude that the behavior $Beh(G)$ of a graph G is the natural join over the behavior of all hyper edges in the graph, while the behavior of the hyper edges is determined by the external behavior of their defining graphs, cf. equation A.12 and A.15, so:

$$Beh(G) = \bowtie_{e \in E_G} Beh(e) = \bowtie_{e \in E_G} (Beh(g_G(e)) \infty h(e)) \tag{A.17}$$

We are now able to derive the behavior of a graph from the behavior of the defining graphs of its hyper edges. One task that remains, is deriving the external behavior of the graph. Restricting a function f to a set A means removing all tuples $\langle a, b\rangle$ from the function of which a is not an element of A. This is denoted by $f\lceil A$, so:

$$f\lceil A = \{\langle a, b\rangle \mid \langle a, b\rangle \in f \wedge a \in A\} \tag{A.18}$$

The projection $T\lceil A$ of a table T to a set A is then defined by the restriction of all functions in T, so:

$$T\lceil A = \{f \mid \exists g : g \in T \wedge f = g\lceil A\} \tag{A.19}$$

Using this definition, the external behavior of a graph can be derived from the behavior of a graph by projecting the behavior on the union of the input and output nodes, so:

$$Beh_{ext}(G) = Beh(G)\lceil(N_{inp,G} \cup N_{out,G}) \tag{A.20}$$

The preceding definitions and formulas are sufficient to derive the behavior of any graph from the primitive graphs and to derive the correctness of the transformations. For example the correctness of the common subexpression transformation in figure A.2 can be proven as follows:
Let F be a function and let $\langle$b, $x\rangle$ and $\langle$r, $\mathcal{V}$(r)$\rangle$ be tuples in an observation O_J of graph J: and $\langle$b, $x\rangle$, $\langle$p, $\mathcal{V}$(p)$\rangle$ and $\langle$q, $\mathcal{V}$(q)$\rangle$ be tuples in an observation O_K of graph K:. Because F is a function $\mathcal{V}$(r) = $\mathcal{V}$(p) = $\mathcal{V}$(q) = $F(x)$. From this the equivalence of the external behavior is immediate. Notice that the transformation does not hold in case F is a relation.

In most cases the graphs model functions. However, the calculus describes relations, because an observation only prescribes the allowed combinations of data values that are bound to the nodes. This gives us the opportunity to describe non-determinism in terms of output don't-cares. Cycles in a graph are allowed but migth result in a empty behavior as is the case in a graph with two hyperedges modeling $a = b$ and $a = b + 1$.

A.5 Conclusions

We presented a 'correctness-by-construction' design flow based on small local behavior-preserving transformations. The design flow is built on a control data flow graph

based design language that describes a design in terms of hyper graphs. The semantics of the design representation are fully based on a set theoretic calculus and applies the concept of the mathematical representation of tables.
The design language is sufficiently powerful for describing complex systems, though, only for systems that are discrete in time and value.
The design flow starts from a specification in C^{++} in the form of a Kahn model of communicating processes. The channels that model the communications are based on a predetermined C^{++} class. However, in the process description object-oriented concepts cannot be used. The principle of the translation of the specification in C^{++} to our design representation is based on fixed templates; each language construct in C^{++} refers to a unique graph template. We have demonstrated the most important steps in the translation process. Generally, the specification is mapped into a synchronous implementation. For this purpose, the design language supports two models for dealing with time. In the synchronous version, the design data values are described for a particular time instance, so time is implicit. A delay operation (register) is provided to refer to data values from the previous time instance. In the process model version, time is explicit and modeled by semi-infinite sequences. Transformations are provided to switch between both models.
A design platform supporting the design flow described in this paper is under construction. This platform is built around a design database in which the design is stored and which provides all primitive graphs and transformations. Furthermore, the design platform contains a compiler for translating the Kahn model in C^{++} to the design language, a graphic user interface for viewing and building graphs and tools for applying and adding transformations.
In the past many relatively small design experiments based on transformational design have been exercised, be it on a slightly different design language. These experiments have shown the feasibility of transformational design [12–14] and its main advantages: correctness-by-construction and therefore no lengthy simulation runs, freedom to exploit the designers creativity, short design times, flexibility in retargeting the design flow from software implementation to hardware implementation and visa versa, ease of handling complexity and a controllable design flow.

Bibliography

[1] Brock, E.O. de,: Foundations of Semantic Databases. Prentice Hall 1995.

[2] Gajski, D.D: Specification and Design of Embedded Hardware-Software Systems. IEEE Design & Test of Computers, (Spring 1995) 53–67

[3] Huijs, C.: A Graph Rewriting Approach for Transformational Design of Digital Systems. Proceedings 22nd EUROMICRO conference, Prague, Czech Republic, (September 1996) 177–184.

[4] Huijs, C.: Transformational Design of Digital Systems related to Graph Rewriting. Proceedings of the Workshop on Design Methodologies for Microelectronics, Smolenice Castle Slovakia (September 1995) 297–305.

[5] Kahn, G.: The Semantics of a Simple Language for Parallel Programming. Information Processing **74** (Aug. 1974) 471–475,

[6] Kock, E.A. de, Essink G., Smits W.J.M., Wolf P. van der, Brunel J.Y., Kruijtzer W.M.,Lieverse P., Vissers K.A.: YAPI: Application Modeling for Signal Processing Systems. Proceedings 37th Design Automation Conference (DAC), (2000).

[7] Lee, E.A., Parks, T.M.: Dataflow Process Networks. Proceedings of the IEEE **83** (1995) 773–801

[8] Leijten, J.A.J., Meerbergen J.L. van: Stream-communication between real-time Tasks in a high performance multiprocessor. Proceedings of DATE 1998 (Feb. 1998) 125–131, .

[9] Meerbergen, J.L. van, Lippens, P.E.R., Verhaegh, W.F.J., Werf A. van der,: PHIDEO: High-level synthesis for high throughput applications. Journal of VLSI Signal Processing **9** (1995) 89–104.

[10] Mekenkamp, G.E., P.F.A. Middelhoek, P.F.A., Molenkamp, E., Hofstede, J., Krol, Th.: A Syntax based VHDL to CDFG Translation Model for High-Level Synthesis. VHDL International Users Forum (VIUF), Santa Clara, (Feb 1996).

[11] Mekenkamp, Gerhard E. A new approach to VHDL-based synthesis.: Ph.D. Thesis University of Twente (Januari 23, 1998).

[12] Middelhoek, P.F.A.,Huijs, C., Mekenkamp, G.E., Prangsma, E., Engels, E., Hofstede, J., Krol, Th.: A Methodology for the Design of Guaranteed Correct and Efficient Digital Systems. IEEE International High Level Design Validation and Test Workshop (HLDVT96). Oakland, California (November 1996).

[13] Middelhoek, P.F.A., Mekenkamp, G.E., Molenkamp, E., Krol,Th,: A Transformational Approach to VHDL and CDFG Based High-Level Synthesis: a Case Study. Proceedings of CICC 95, Santa Clara, Ca., (May 1995) 37–40,

[14] Middelhoek, P.F.A.: Transformational Design: an architecture independent interactive design methodology for the synthesis of correct and efficient digital systems. Ph.D. Thesis University of Twente (April 3, 1997).

B
Tiny Z Toolkit

B.1 Basic types

$[CHAR]$

$\mathbb{B} ::= \mathtt{0} \mid \mathtt{1}$

B.2 Sequence operations

$[X]$

$$\_ \leftarrow\!\!\square\ \_ , \_\ \square\!\!\rightarrow\ \_ : \mathbb{N} \times \operatorname{seq} X \longrightarrow \operatorname{seq} X$$

$$\forall n : \mathbb{N}; x, x' : \operatorname{seq} X \mid n \leq \#x \bullet$$
$$n \leftarrow\!\!\square\ x = x' \Leftrightarrow (x' \operatorname{prefix} x \land \#x' = n) \land$$
$$n\ \square\!\!\rightarrow x = x' \Leftrightarrow (x' \operatorname{suffix} x \land \#x' = \#x - n)$$

$[X]$

$$write : \mathbb{N} \times \operatorname{seq} X \times \operatorname{seq} X \longrightarrow \operatorname{seq} X$$

$$\forall offset : \mathbb{N}; segment, list : \operatorname{seq} X \mid \#list \geq offset + \#segment \bullet$$
$$write(offset, segment, list) = (offset \leftarrow\!\!\square\ list)^\frown$$
$$segment^\frown (offset + \#segment\ \square\!\!\rightarrow list)$$

B.3 Projections

$$
\begin{array}{|l}
\hline
[X, Y, Z] \\
\hline
\pi_1 : X \times Y \times Z \longrightarrow X \\
\pi_2 : X \times Y \times Z \longrightarrow Y \\
\pi_3 : X \times Y \times Z \longrightarrow Z \\
\hline
\forall\, x : X; y : Y; z : Z \bullet \\
\quad \pi_1(x, y, z) = x \land \\
\quad \pi_2(x, y, z) = y \land \\
\quad \pi_3(x, y, z) = z \\
\hline
\end{array}
$$

Bibliography

[1] Michael Barr and Charles Wells. *Category theory for computing science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[2] Charles Donnelly and Richard Stallman. *Bison - The Yacc-compatible Parser Generator*. Free Software Foundation, Boston, MA, USA, December 2004. Manual for Bison version 2.0, available from `http://www.gnu.org/bison/gnu`.

[3] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. John Wiley, 2002.

[4] Corrie Huijs. A graph rewriting approach for transformational design of digital systems. In *Proceedings 22nd EUROMICRO conference, Prague, Czech Republic*, pages 177–184. IEEE Computer Society Press, September 1996.

[5] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1998.

[6] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. pub-ISO, pub-ISO:adr, December 1999.

[7] Jonathan Jacky. *The way of Z: practical programming with formal methods*. Cambridge University Press, New York, NY, USA, 1996.

[8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[9] W.E.H. Kloosterhuis, M.M.R. Eyckmans, T. Krol, O.P. McArdle, and W.J.M. Smits. Sil-2 language report. Technical Report PR-5.2, Sprite Consortium, September 1993.

[10] Thijs Krol and Bert-Steffen Visser. High-level synthesis based on transformational design. Internal report for the Embedded Systems group of the University of Twente, included as appendix A.

[11] Stephen Prata. *C++ Primer Plus*. Waie Group Press, 1998.

[12] J.Mike Spivey. *The Z Notation: a reference manual*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992. Available from `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.

[13] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, 1976.

[14] ioccc. The international obfuscated C code contest.

[15] David A. Watt. *Programming language syntax and semantics*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1991.