FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

MPI-Model-Checking in C-Programmen

Christian Iwainsky

Diplomarbeit

MPI-Model-Checking in C-Programmen

Christian Iwainsky

Diplomarbeit

Aufgabensteller:	Prof. Dr. C. Pflaum
Betreuer:	Prof. Dr. C. Pflaum
Bearbeitungszeitraum:	28.2.2007 – 27.8.2007

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 27. August 2007 .....

Abstract

English

The message passing interface, usually called MPI, is the de facto standard in scientific parallel programs for specifying the communication behavior. Even though MPI greatly assists the programming of parallel programs with easy to use communication primitives, the development is not very intuitive and prone for error.

Model checking is a technology to verify a given model for correctness. However, one first has to obtain a suitable model before a verification is possible, which is not a trivial task. To construct the communication model of a C-program using MPI one not only has to know the program but also has to know at least one model checking system and its input method. Since many who use MPI have no such background in model checking this software development technology is hardly used.

This work presents an automatic process, that automatically extracts the MPI communication structure from a C-program and generates a model description for the MPI-SPIN [20, 21] model checker. In contrast to existing work, like the SLAM-project [4], ModEx [11] and the Java Path Finder [26], the developed process does not require any user input and automatically performs the necessary model abstraction. The abstraction and extraction process uses the ROSE-framework [17, 19] as a platform for transformation and analysis of the source program.

The presented automatic model extraction has been applied to a few sample programs. The thereby generated models were then tested for their informational value regarding error detection of the used communication scheme. In general, the extraction process and the following verification were successfully, though in some cases the extracted model was over-abstracted. This over-abstraction caused the extracted models to exhibit faulty behavior, which the original program did not have. Theses so called false-negatives are a known problem in a model checking community. The reason for these false-negatives, and thereby the limiting factor for model checking, is the necessity to abstract data which can induce non-determinism into the model.

Unfortunately the extraction process could not be tested on generic MPI-programs, because MPI-SPIN does not provide communication groups in its current version. Also some minor stability problems with some of ROSE's non final analyses and transformations prevented extensive testing of MPI-calls nested in functions.

German

Das *message passing interface*, kurz MPI, ist der allgemeine Standard für wissenschaftliche Anwendungen, um die Kommunikation zu implementieren. Trotz der Tatsache, dass MPI die Programmierung von parallelen Programmen mittels leicht zu verwendender Kommunikationsprimitiven unterstützt, ist die Entwicklung von parallelen Programmen fehlerträchtig und nicht sehr intuitiv.

Model Checking ist eine Technologie, mit der man ein gegebenes Modell auf Korrektheit überprüfen kann. Jedoch besteht die Schwierigkeit darin, das Modell zu erstellen, bevor man eine Überprüfung durchführen kann. Um das Kommunikationsmodell eines C-Programmes das MPI verwendet zu bilden, muss man nicht nur das Programm sondern auch ein Model-Checking-System und seine Eingabemethodik kennen. Da viele MPI-Anwender kein entsprechendes Fachwissen haben, wird diese Entwicklungstechnologie jedoch kaum genutzt.

Diese Arbeit stellt einen automatischen Prozess vor, der selbständig die MPI Kommunikationsstruktur aus einem C-Program extrahiert und eine Modellbeschreibung für den MPI-SPIN-Model-Checker [20, 21] erstellt. Im Gegensatz zu existierenden Systemen, wie das SLAM-project [4], ModEx [11] und dem Java Path Finder [26], benötigt der entwickelte Prozess keinerlei Eingriff des Anwenders und führt automatisch die notwendigen Abstraktionsschritte aus. Für die Abstraktion und Extraktion wird das ROSE-System als eine Plattform für Transformationen und Analysen des Quellprogramms genutzt.

Die in dieser Arbeit vorgestellte Modell-Extraktion wurde an einigen Programmen auf Korrektheit überprüft. Die hierbei erstellten Modelle wurden im Anschluss bezüglich ihrer Aussagekraft in Bezug auf Fehlererkennung in der Kommunikationstruktur getestet. Im Allgemeinen war der Extraktionsprozess und die anschließende Fehlererkennung erfolgreich. Jedoch wurde in einigen Fällen eine zu starke Abstraktion des Modells festgestellt. Eine solche Überabstraktion führt zu einem fehlerhaften Verhalten des extrahierten Modells, welches nicht dem Verhalten des Originalprogramms entspricht. Diese sogenannten *false-negatives* sind ein bekanntes Problem im Bereich des Model Checkings. Der Grund für diese *false-negatives* liegt in der Notwendigkeit, Daten zu abstrahieren. Dies führt zu Nicht-Determinismus in einem anderemfalls deterministischen Modell und ist somit der begrenzende Faktor für *Model Checking*.

Leider konnte im Rahmen dieser Arbeit der Extraktionsprozess nicht für allgemeine MPI-Programme getestet werden, da MPI-SPIN in seiner aktuellen Version keine Kommunikationsgruppen unterstützt. Weiterhin verhindern Stabilitätsprobleme von einigen sich im Entwicklungsstadium befindenden Analysen und Transformationen aus ROSE ausführliche Tests von MPI-Aufrufen, die in Funktionen eingebettet sind.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Big calculations and scientific programs require more and more powerful computer systems. Even though technological progress has lead to an increase in computing power, storage capacity and memory size for a single system, the demand for faster and larger systems continues to grow. Cluster computing is a way to counter this hunger for more computing capacity by combining the computation power of multiple computers to solve a single problem. A computer cluster is a set of computers, called *nodes*, which have been connected in some way, usually by a computer network, to combine their resources, like computing power and memory.

Cluster systems usually are classified by their memory architecture. The common high performance architectures are shared memory systems, distributed memory systems and distributed shared memory systems. In a shared memory system each node has direct access to all memory, which is a very expensive and complicated setup. On the other hand for a distributed memory system each node has only access to its own memory and must in some form communicate with the other nodes to exchange data. This corresponds to a system of ordinary computers without special soft- and hardware. The distributed shared memory architecture is in some aspects an extension of the distributed memory system. In such a system each node has its own local memory, which it actively shares with the other nodes, therefore creating a distributed, but shared memory with a common address space. This is accomplished either in software or with special hardware by intercepting non local memory requests and forwarding these to the corresponding node, which then executes the memory access on behalf of the requesting node. Today cluster systems usually are distributed memory systems, which consist of many single computers connected by a high performance network.

Even though the idea for parallel computing is fairly simple, the development of parallel applications is a difficult and demanding process and often requires special knowledge. For a program to use multiple computers for a single computation it must be specially written or adapted to utilize the distributed resources and to perform the necessary communication. This parallelization process is often very tedious, since bugs are common and very difficult to detect, to reproduce and in the end to fix.

This difficulty originates in the concurrent behavior of such systems and the often absence of a common precise global clock. An alternative to a global clock is to synchronize the local clocks, where it is possible to achieve synchronization down to milliseconds [24]. Nevertheless, this is not sufficient in many cases, because even in that small amount of time todays CPUs execute hundreds to thousands of instructions and because of this one can not call such nodes synchronized.

Typical problems that one faces when developing a parallel application are *racing conditions*, *deadlocks* and *lifelocks*. A racing condition describes the situation when multiple processes or threads compete without success for an exclusive resource, like a counter or a barrier. A deadlock describes, that computations halts due to some undesired behavior, where a lifelock depicts a behavior where the system is still executing, but does not make a desired progress.

Furthermore errors can usually not be reproduces directly without some effort, because parallel systems very rarely exhibit the exact same execution twice, such that the precise sequence of messages on the network and of instructions on the nodes are the same. Next to the fact that the different processes can not be started perfectly synchronous, this can also be explained by operating system intermittently influencing the processes, like performing swapping operations or even process scheduling.

Additionally it is very challenging to utilize the full power of a cluster system. Ideally, if one runs a program in parallel on N machines, one would like the execution of the program to take only the N-th fraction of the time, compared to the time that the program would take on a single machine. In order to try to come close to this ideal parallel speed up, which is the best serial wall time divided by the best parallel wall time, programmers often employ complex algorithms and communication schemes.

1.1 MPI

An established way to implement this communication for parallel applications is message passing. With message passing each process having its own process environment is independent from the other processes and communicates by sending explicit messages, which usually contain data or control information. This paradigm still requires the programmer to implement large parts of the communication by himself, like accessing the networking interface of the operating system and implementing the communication startup and error recovery. To ease the development and implementation overhead for parallel applications the *message passing interface* standard [14], short *MPI*, was developed.

MPI is a system and platform independent message passing standard that facilitates the development of parallel applications and libraries. It defines syntax and semantics of core functions necessary to implement communication for parallel computing. Additionally MPI defines advanced features helpful for developing parallel applications like user defined data types, persistent communications ports, synchronization methods, collective operations and scoping mechanisms for communication. The interface also provides a virtual topology which enables addressing of nodes independent from the actual network architecture.

Most MPI implementations provide an API[1] that is available in form of a library. These libraries usually have been optimized for a specific architecture and operating

[1] Application Interface

system to provide the best performance and scalability possible. Often customized MPI implementations are available for different shared memory, distributed memory and non uniform memory architectures, taking advantage of the specific hardware environment.

Due to these advantages in portability and scalability MPI is today widely used in scientific computation. The interface is available to almost any language that supports routine libraries, like FORTRAN, C and C++. Nevertheless one must not forget, that, despite of MPI's acceptance, it is primarily only an interface description and not a language extension.

One of the drawbacks of MPI is, that there is no inherent debugging support which possibly leads to MPI applications to contain communication errors that may be very difficult to detect. An example for such an error can even be found in popular MPI literature. For instance, it has been shown in [20] that in example 2.17 from [22] because of the communication, some variables could overflow, which could lead to undesired behavior.

Such errors in the communication are difficult to find and, if existent, do even not often actually occur during the executing of a program. For example, an application that works on a cluster with a small number of computers may block or crash on a larger cluster. A possible cause for this can be that for the small cluster the messages arrive in their desired order, but for the larger system because of the increased communication the network is getting close to its bandwidth limit, therefore causing messages to arrive out of order. In general, the reasons for these errors may be very difficult to find and to resolve, possibly wasting a lot of time and effort in the process.

1.2 Model checking

With this in mind, one might wish, that there would be an application or process that could automatically decide if a given program would always work the way it supposed to work, without having to rely on statistic and empirical testing. Unfortunately this can not be done for arbitrary programs. For example it was profen by Turing in 1936, that it is impossible to automatically verify if any arbitrary given program (Turing program) terminates [25]. Even though arbitrary programs cannot be verified, it is possible to verify properties for specific programs or abstract models, which are more design like and do not represent implementation details.

This verification process, or more precisely the process to check if a given system description satisfies a logical formula, is called model checking. Model checking has been used in protocol and hardware design to find design flaws in different systems. Besides the academia model checking has been utilized by NASA, that used this technique to check the control systems of a space probe [10], and by different industrial companies [2].

In the beginning model checking was very limited, since it is very expensive in terms of memory consumption and computing power. In recent years the capabilities of model checkers has greatly increased due to continued research and increase in memory capacity

and machine performance. A model checking tool usually accepts a model description and properties for a given model and then verifies, if the model satisfies the properties or not. These properties can be very different, but in general can be categorized as safety or liveness properties. A safety property states that something bad does not happen, like a deadlock or a program crash, where a liveness property states that something good happens, like progress in the computation. The precise definition can be obtained from [6]. Most modern automatic model checkers transform the input model in some form of an automaton and use temporal logic to verify the given properties on that automaton. The limiting factor for this process is the size of the model representation, which is called the *state explosion*, and the storage and processing requirements during verification, which usually grow exponentially.

1.3 Outline

One of the aspects that hampers the widespread use of model checking is that it is rather difficult to acquire a model of a given problem. In order to get a model for a specific problem one has to analyse a program or the mostly prose based specification for a system and manually generate a model for that system. At the same time one has to determine the properties that are of interest and relevance and how to represent those properties within the model. In perspective of this situation the goal of this work is to develop a new fully automatic process to extract the MPI communication model from a given C program without any user interaction. First a short, more profounded introduction of model checking is given. Later the design decisions and general implementation details will be described. Lastly the extraction process will be evaluated for its use and an outlook of future developments is given.

2 Fundamentals of model checking

This chapter will give a short introduction to the ideas and theory behind model checking in order to better understand, what model checking is and what its capabilities are. This knowledge is also necessary to understand the capabilities and limitations of the used model checking system and the model extraction process described later in this work.

As already stated in the introduction, model checking is a process to verify if a given model satisfies a set of properties. So far it has not been explained how one accomplishes this. Since the focus of this work is on dynamic concurrent behavior one must find a precise and efficient way of representing the model in a formal way.

2.1 Automata

The most basic primitives that one needs to describe almost any process are sequences of actions or statements, jumps within these sequences and some method of conditional branching. In this case prose language itself is of little use, since it is notoriously ambiguous and rather difficult to precisely argue about, without going through a lot of effort to specify the implicit meaning of such statements. This is even more so, when trying to automatically argue about such statements. The concept of *automata* is reasonably well suited to act as a formal representation for such systems without too much limiting expressiveness of the model [6].

An automaton, also in some cases a state machine, is a set of labels ($\mathcal{S}$), called *states*, a set of transition relations (δ) between states, a set of start states (s_0),a set of input (Σ) symbols and a set of end states ($\mathcal{F}$) (Def. 1).

Definition 1:
$\mathcal{A} = (\Sigma, \Gamma, \mathcal{S}, s_0, \delta, \mathcal{F}, \omega)$ *with*
$\mathcal{S}$ *a finite set of states*
Σ *a finite set of input labels*
δ *a set of transitions of the form* $\delta \subseteq \mathcal{S} \times \Sigma \mapsto \mathcal{S}$
$\mathcal{F}$ *the set of end-states.*

This basic automaton has some special final states ($\mathcal{F}$), which indicated that a specific series of symbols has been recognized or accepted. Such an automaton is called *recognizing automaton*. Figure (Fig. 2.1) shows an example for such an automaton. An related representation, which is also often used in model checking is the *kripke structure*, which is an automaton without the input symbols (Σ) and the final states ($\mathcal{F}$) but extended

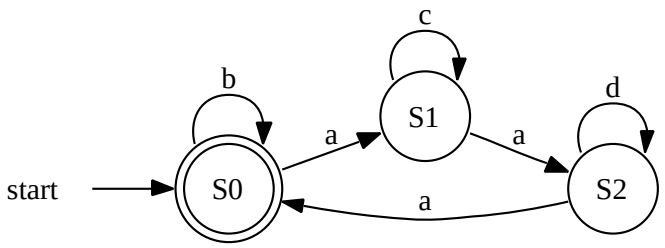

Figure 2.1: A simple automaton with three states and input labels a,b,c and d

with a special interpretation function ($\mathcal{L}$). There are many further different variations of the automaton concept in computer science, like automata with output. These variation are however not relevant to this work, though the mechanisms described below still apply.

Additionally to being a good representation for a model automata have a second advantage. It is in formal terms easy to combine multiple automata to form a combined automaton, thereby enabling a representation of concurrent behaviour. This combination process is defined by the automata product. Without loss of generality for any automaton representation, this will be explained for a recognizing automaton.

The product of n automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ with $\mathcal{A}_i = (\Sigma_i, \mathcal{S}_i, s_{0,i}, \delta_i, \mathcal{F}_i)$ and $1 \leq i \leq n$ is written $\mathcal{A}_P = \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ and defined by:

Definition 2:
$\mathcal{A} = (\Sigma_P, \mathcal{S}_P, s_{0,P}, \delta_P, \mathcal{F}_P)$ *with*

$$
\begin{aligned}
\Sigma &= \textstyle\prod_{1\leq i\leq n}(\Sigma_i \cup \{\epsilon^1\}) \\
\mathcal{S}_P &= \mathcal{S}_1 \times \ldots \times \mathcal{S}_n \\
s_{0,P} &= (s_{01}, \ldots s_{0n}) \\
\delta &= (q_1, \ldots, q_n), (e_1, \ldots, e_n) \rightarrow (q'_1, \ldots, q'_n) | (\forall i, e_i = \{\epsilon\} \wedge q'_i = q_i) \vee \\
&\quad \vee (e_i \neq \{\epsilon\} \wedge q'_i \in \delta_i) \\
\mathcal{F} &= \mathcal{F}_1 \times \ldots \times \mathcal{F}_n \ .
\end{aligned}
$$

For this product, each single automaton within the combined automation is in the beginning still independent of the other automata, because with the *no operation* or *empty word* symbol, written (ϵ), every automaton has the ability to remain in its current state. In the case no restrictions apply to the composition of the input-pairs this product is called *unsychronized*.

If the composition of the set of input-pairs is restricted, also called synchronized, thereby limiting the available transition to the subset $\sigma_{Sync} \subset \sigma$, then one obtains a *sychronized* automaton product (Def. 4).

[1] ϵ represents the empty symbol

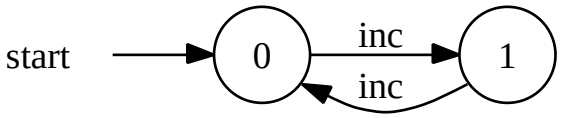

Figure 2.2: A modulo two counter automaton

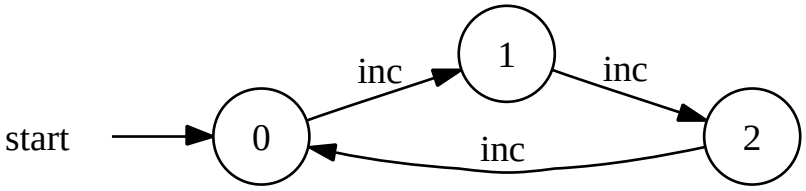

Figure 2.3: A modulo three counter automaton

Definition 3:
$Sync \subseteq \prod_{1 \leq i \leq n} (\Sigma_i \cup \{\epsilon\})$.

With the set *Sync*, (Def. 3), the transitions δ_{Sync} for the synchronized automaton are defined by (Def. 4).

Definition 4:
$\delta_{Sync} = (q_1, \ldots, q_n), (z_1, \ldots, z_n) \rightarrow (q'_1, \ldots, q'_n) \in \delta \ \textit{with} \ (z_1, \ldots, z_n) \in Sync$

One can understand this difference by studying the product of a modulo two $\mathcal{A}_1$ (Fig. 2.2) and a modulo three $\mathcal{A}_2$ (Fig. 2.3) counter automaton. These automata operate with incrementation operation, called *inc*, and are defined by the definition:
$\mathcal{A}_1 = (\{0,1\}, \{inc\}, \{(0, inc) \rightarrow (1), (1, inc) \rightarrow (0)\}, 0, \oslash)$
$\mathcal{A}_2 = (\{0,1,3\}, \{inc\}, \{(0, inc) \rightarrow (1), (1, inc) \rightarrow (2), (2, inc) \rightarrow (0)\}, 0, \oslash)$
If one generates the asynchronous product of $\mathcal{A}_1$ and $\mathcal{A}_2$, allowing all combinations of ϵ and *inc*, one obtains the automaton $\mathcal{A}_{async}$:

$$\mathcal{A}_{async} = (\{(0,0),(0,1),\ldots,(1,3)\}, \{(inc,inc),(\epsilon,inc),(inc,\epsilon),(\epsilon,\epsilon)\},$$
$$\{((0,0),(inc,inc)) \rightarrow (1,1)), ((0,1),(\epsilon,inc)) \rightarrow (0,1),\ldots,\}, (0,0), \oslash)$$

It is obvious that the number of transitions is pretty large, 24 in this case, compared with the number of transitions of $\mathcal{A}_1$ or $\mathcal{A}_2$. Additionally every state for this automation is freely accessible and no specific behavior of $\mathcal{A}_{async}$ can be specified.

On the other hand the number of traditions drastically reduces if one selects a *synchronized* subset of input symbols. For example, if the input only consists of the paired

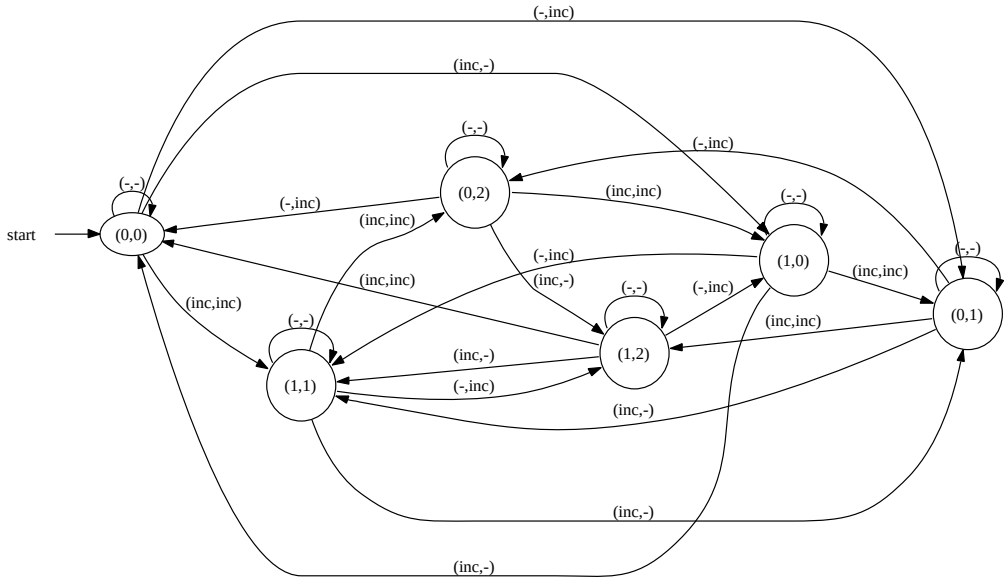

Figure 2.4: The asynchronous product of $\mathcal{A}_1$ and $\mathcal{A}_2$

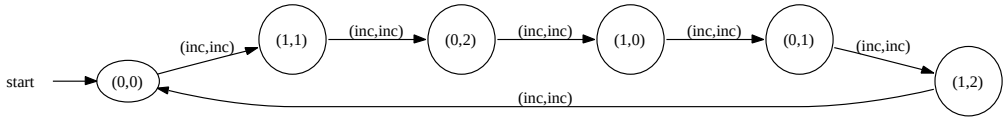

Figure 2.5: A synchronized product of $\mathcal{A}_1$ and $\mathcal{A}_2$

incrementation of each automation $E = (inc, inc)$, then the number of transitions drastically reduces and the product is a modulo 6 counter:

$$\mathcal{A}_{sync} = (\{(S0,S0),(S0,S1),\ldots,(S1,S3)\},\{(inc,inc)\},\\ \{((S0,S0),(inc,inc)) \rightarrow (S1,S1)),\ldots,\\ ((S1,S2),(inc,inc)) \rightarrow (S0,S0)\},(S0,S0),\oslash)$$

In this case also all states are accessible from the start-state, but only in one possible order (Fig. 2.5). It should be noted, that it is generally not the case, that all states of the combined automaton are accessible beginning from the start state, but one often obtains multiple independent sets of automata. For a more elaborate example one may refer to [6].

The interesting point with synchronization is, that one can model communication between the automata by correctly restricting the synchronized input. The transition $(A_v, B_x), (Send, Receive) \rightarrow (A_w, B_y)$ is an example for such a sychronized communication event for the automata A and B. In this case the synchronization is the pairing of the send-symbol the receive-symbol.

With such a synchronization mechanism one can even model a more complicated communication behavior, like buffered channels or asynchronous message passing. For this, instead of directly coupling the different automata, one implements an additional automaton to describe the behavior of the communication channel. By connecting this message-channel-automaton to the communicating automata and by modifying the synchronization set, complex communication behavior can be described.

So far, it has been described, how one can model concurrent behavior and control-flow constructs with automata, but in order to represent a model powerful enough to study more than trivial systems, this basic idea of an automaton has to be extended to contain a notion of variables.

Typically a variable is a placeholder for something else, like numeric values or symbols. To use a variable together with automata, one could store the value of that variable implicitly within the states, thereby creating a set of extended states, with only the value associated with each state as a distinction. For example, if one wanted to model, that a variable at a state V has the value of 5, a new state $V5$ would be added to the original automaton. This representation corresponds to the product of the original automaton and an automaton representing the numeric values of $\mathbb{N}$. It is clear, that any variable can be expressed by an automaton with a number of states equivalent to the number of different values that the variable can represent. Unfortunately this representation is rather unpractical.

Because the explicit formal representation is very expensive in memory consumption, a more space saving approach models a variable by folding the states of the variable to form a common state. This also requires the transactions to be merged and a new representation of the implicit numeric calculations and conditions [6]. Typically this is represented with *guards* that control if a given transition can be executed and deterministic atomic actions describing the calculations for that transition.

This folding can also be applied to multiple variables. In the end one obtains an automaton, that represents the control flow of the model directly in states and transitions and stores variables used in the model at the labels. This set of variables at a single state is called called *state-vector*. Such a folding can be seen in figures (Fig. 2.1) and (Fig. 2.6). The transitions from figure (Fig. 2.1) is folded to construct the folded tansitions of figure (fig. 2.6). The if on the transition only express executability of a transition and does not add additional further transitions to the automaton.

2.2 Temporal logic

With the possibility to express a dynamic model in form of a synchronized automaton, it remains to define how to formally, logically and precisely argue about such an automaton.

Without a special adapted logic, one has to fall back to use primary logic for a verifcation of such a system. For example, one would use a first order formula to expresses temporal behaviors by using functions and variables of time to express temporal order-

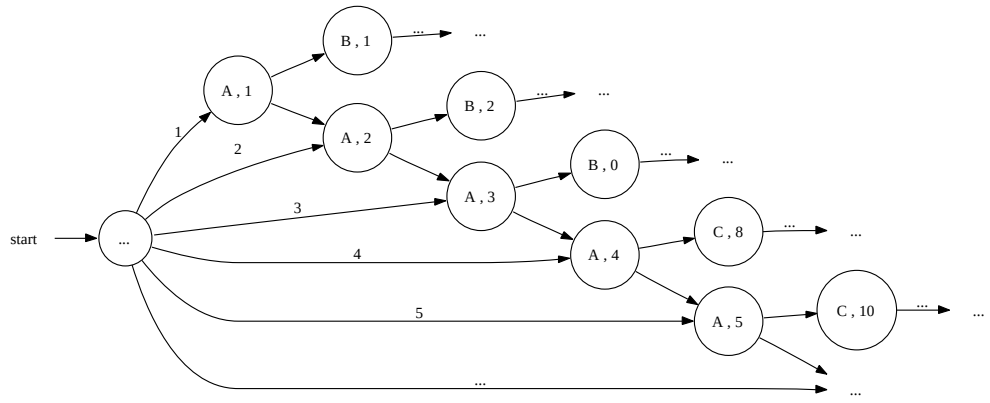

Figure 2.6: Excerpt of an unfolded automaton

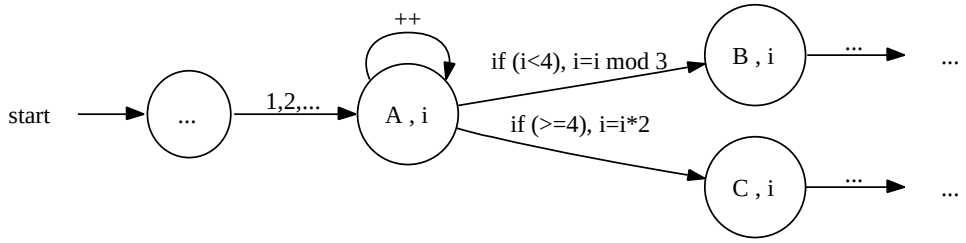

Figure 2.7: Excerpt of a folded automaton corresponding to figure (Fig. 2.1)

ing. The expression $\exists t, i \in \mathbb{N} : \exists t^{'}, i(t^{'}) = 0; t^{'} > t$ is an example for such a formula. It describes that the variable i will have the value 0 sometime in the future. However, this example does neither state that the condition has to be fulfilled in the same execution nor when t or $t^{'}$ occur. For model checking this representation is rather unsuitable, since one has to define the model of time and its correspondence to transitions and different executions. Additionally, one would have to explicitly model the temporal behavior, using this more complex form.

To avoid having an explicit representation of time and to make reasoning of dynamic behavior easier *temporal logic* was defined. Temporal logic requires time to be seen as a sequence of states or as discrete points of time and defines operators to express temporal behavior relative to these steps, using *atomic propositions*, *boolean operators*, *temporal operators* or combinations there of.

As with primary logic, *atomic propositions* are boolean statements, that either evaluate to true or false, like the comparison of variables. The well known *boolean operators* are the *true* and *false* values, the *negation* ($\neg$), the *and*-operation ($\wedge$), the *or*-operation $\vee$, the *implication* ($\rightarrow$) and the *dual implication* ($\leftrightarrow$).

Name	Text form	Symbolic form
until	ϕUψ	$\phi\mathcal{U}\psi$
next	Xϕ or Nψ	$\circ\phi$
finally	Fϕ	$\Diamond\phi$
globally or guarantee	Gϕ	$\Box\phi$
all or always	Aϕ	$\mathcal{A}\phi$
exists	Eϕ	$\mathcal{E}\phi$

Table 2.1: Overview of temporal symbols

2.2.1 Temporal operators

Additionally to those basic boolean operators, the temporal logic defines *temporal operators* to express future propositions. The operators are the binary *until* (U), the unary *next* (X), *finally* (F), *globally* (G) and the path quantifiers *always* (A) and *exists* (E). The until operator $\phi U\psi$ states, that the temporal formula ϕ at least holds until ψ becomes true but requires that ψ must become true at some later point. The unary next $N\phi$ states, that for the next time step the formula ϕ holds. The operator finally $F\phi$ states, that at least once ϕ becomes true at some future point, whereas the globally operator $G\phi$ represents that for the current state and for each of the future states ϕ has to hold. The operators globally and finally are complementary and is expressed with $G\phi = \neg F\neg\phi$. This reads 'It will not happen once or more that ϕ does not hold'[6]. Additionally one uses often a weak until $\phi W\psi$, which has the same meaning as until but without the requirement that ψ has to eventually occur [12].

The second addition to the basic logic is the concept of path quantifiers which express possibilities of future behavior, without introducing direct probabilities. There are two unary path operators, the all operator A, which is also called always, and the exists operator E. The operator always $A\phi$ expresses that for all futures, more precisely future states, ϕ will hold at some arbitrary time. This point of time, when ϕ holds, does not have to be precisely the same state, for all possible executions. In contrast $E\phi$ states, that it is possible for ϕ to become true at some future time, but it does not have to.

Of course temporal logic also allows nesting of operators and parenthesis to group sub-formula. Two noteworthy nestings are $FG\phi$, also sometimes written as $G^{\infty}\phi$, with the meaning of 'all the states form a state onward have to satisfy' [6, 12] and $GF\phi$, written as $F^{\infty}\phi$, with the meaning of 'always there will be a state that satisfies' [6, 12].

In addition to the text form one also encounters a symbolic form of these temporal operators in literature. A quick overview of the these operators and their respective representation can be obtained from table (Tbl. 2.1).

2.2.2 CTL and LTL

Two of the most prominent representations of temporal logic used for model checking are the computation tree logic, short CTL, and the propositional linear temporal logic, abbreviated PLTL or just LTL. Both logics were initially defined separately from each other and expressed different subsets of generic temporal logic. Later both logics were combined to form the more general CTL*. The computation tree logic is a subset of the general temporal logic and is defined by requiring that the state-operators (U,X,F,G) always have to be used in conjunction with a path-operator (A,E). The resulting temporal operators are A(...)U(...), AX, ..., EG. A temporal expression using only these combined operators is called a *state-formula*.

The LTL, in contrast does not contain the path operators A and E. This limits the expressiveness of temporal formula to general statements of behavior and does not provide the means to express possible futures. Therefore in LTL a formula either holds for all possible futures or not at all.

These restrictions on LTL and CTL are originally rooted in the process, how these logics are verified, which will be explained shortly.

2.3 Model checking

With CTL* and automata one can now specify a formal process to verify if a model meets its specification. This can be rephrased to the question, if a given automaton $\mathcal{A}$ satisfies the temporal formula ϕ.

This question can be answered by verifying for each possible execution of $\mathcal{A}$, called path σ, that the CTL*-formula ϕ holds. In the case of automata, a path is defined as the chronological ordered set σ of states $\sigma_{i\geq 0}$, that the automaton traverses during one of its executions:

Definition 5:
$\sigma : \sigma_0, \sigma_1, \ldots, \sigma_n | \sigma_i, s \rightarrow \sigma_{i+1}, \sigma \in \mathcal{S}, s \in \Sigma, \sigma_n \in \mathcal{F}.$

A given temporal formula ϕ is said to hold for the sub-path σ_i at time i, written as $\sigma, i \models \phi$, if all the sub formula of ϕ hold in regard to the following definitions (Def. 6), (Def. 7) and (Def. 8)[6].

Definition 6:

$$\begin{array}{lcl} \sigma, i \models \mathcal{P} & \iff & \mathcal{P} \in l(\sigma_i) \\ \sigma, i \models \neg\phi & \iff & \neg\sigma, i \models \phi \\ \sigma, i \models \phi \wedge \psi & \iff & (\sigma, i \models \phi) \wedge (\sigma, i \models \psi) \end{array}$$

Definition 7:

$$\begin{array}{lcl} \sigma, i \models X\phi & \iff & i < |\phi| \wedge \sigma, i+1 \models \phi \\ \sigma, i \models F\phi & \iff & \exists j, i \leq j \leq |\sigma| \wedge (\sigma, j \models \phi) \\ \sigma, i \models G\phi & \iff & \forall j, i \leq j \leq |\sigma|; \sigma, j \models \phi \\ \sigma, i \models \phi U \psi & \iff & \exists j, i \leq j \leq |\sigma|; \sigma, j \models \psi \wedge \\ & & \forall k, i \leq k \leq |\sigma|; \sigma, k \models \phi \end{array}$$

Definition 8:

$$\begin{array}{lcl} \sigma, i \models E\phi & \iff & \exists \sigma', \sigma(0) \ldots \sigma(i) = \sigma'(0) \ldots \sigma'(i) \wedge \sigma', i \models \phi \\ \sigma, i \models A\phi & \iff & \forall \sigma', \sigma(0) \ldots \sigma(i) = \sigma'(0) \ldots \sigma'(i); \sigma', i \models \phi \ . \end{array}$$

If the given formula applies for all executions of the automaton, then the automaton satisfies the temporal formula (Def. 9).

Definition 9:

$\mathcal{A} \models \phi \iff \sigma, 0 \models \phi \ \forall \ \sigma \ \textit{of} \ \mathcal{A}$

In this context the concept of the path-operators becomes more clear. The all operator states that for a given beginning fragment of a path $\sigma_{0\ldots i}$ all paths of the form $\sigma_{0\ldots i}\sigma_{i+1\ldots}$ ultimately have to satisfy ϕ, whereas the exists-operator requires at least one of these paths to satisfy ϕ.

Having defined the interaction of automata and temporal logic, one can construct an verification system. Given an automaton and a temporal formula in a suitable representation such a system can determine, if the model satisfies the property. The implementation of this method usually differs fundamentally whether the formula is specified in CTL or LTL, because certain inherent properties of the temporal formula can be exploited for the verification.

2.3.1 Model checking CTL

The idea of automatically verifying a CTL-formula is based on the fact, that one can only specify reachability of states in CTL [6]. Therefore one can exploit, that it is possible to conclude at each single state of the automaton, if the temporal formula, or a sub-formula, holds by solely inspecting the direct successor states. This becomes clear, if one looks at the possible temporal operators. For instance EFϕ is satisfied for a given state, if Fϕ holds for the current state or EFϕ holds for at least one of the immediate successor states, which is directly implied by the exists operator. If it is unknown, if EFϕ holds for a successor state, that state is checked in a recursive manner. Similarly, the all operator can only hold for a given state, if its expression holds for all the direct successor states or for the state itself. By introducing a marker, that states that a sub formula ϕ_i or atomic property $\mathcal{P}$ holds for a state, one even does not have to recursively visit each state for each formula. Instead each state is marked for all atomic propositions and then the sub-formula are propagated in the automaton until a fixpoint has been achieved. Once

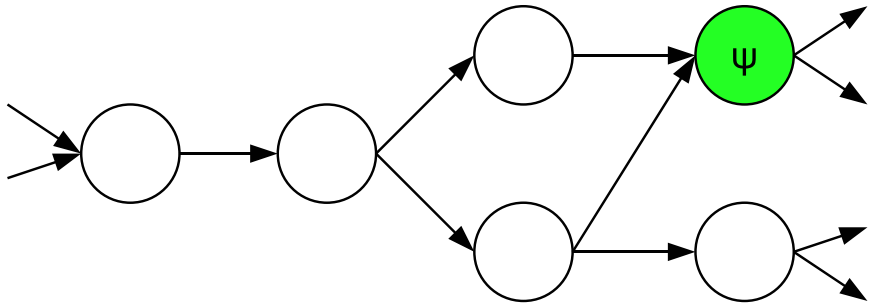

Figure 2.8: Automaton with property Ψ

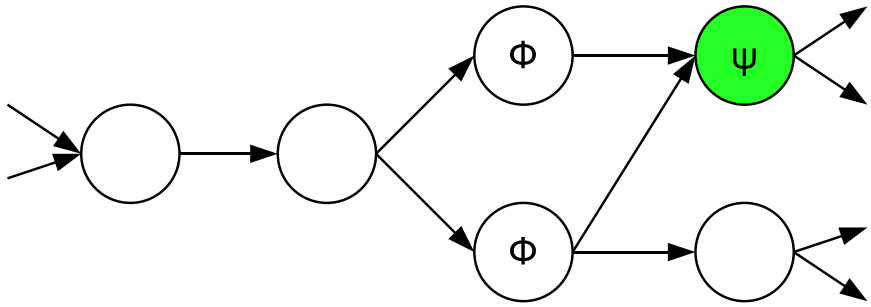

Figure 2.9: Propagation of $CTL\phi = EF(\Psi)$

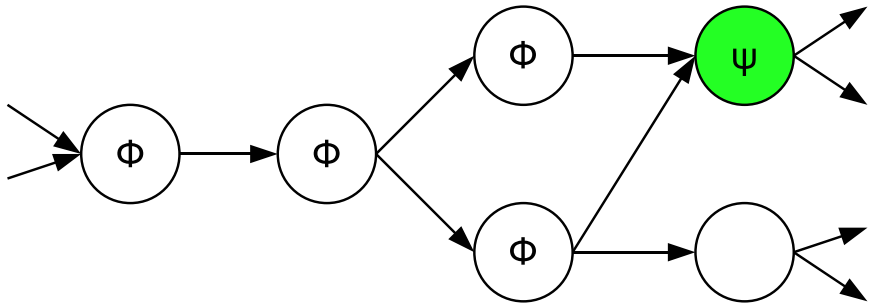

Figure 2.10: Finished marking, Φ holds for initial state

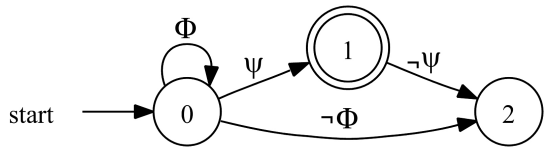

Figure 2.11: Recognizing automaton for $\Phi U \Psi$

complete,the automaton satisfies Ψ, if all start states have been marked with Ψ. Figures (Fig. 2.8),(Fig. 2.9) and (Fig. 2.10) show, how this propagation technique is applied for an automaton. The atomic proposition Ψ holds in (Fig. 2.8) and the statement $\Phi = EF\Psi$ is propagated as shown in (Fig. 2.9) and (Fig. 2.10). In (Fig. 2.10) it can be decided, that the formula holds, because Φ is marked at the start-state.

It should be noted that the given approach can only be applied to finite automata, since it is not possible to mark an infinite set of states.

2.3.2 Model checking LTL

Opposing to the method presented for CTL, LTL-formula can not be decided at the state level. This is specially obvious with expressions of the form GF, which describes behaviors of infinite paths. And since it is impossible to decide path properties at a state level, LTL can not be decided using the same approach as CTL.

To still be able to verify a LTL-formula one has to resort to the fact, that every LTL formula can be accepted by a recognizing finite automaton [6]. For example, the formula $\Phi U \Psi$ corresponds to automaton (Fig. 2.11), which then can be strongly coupled to the existing model automaton. Such an LTL automaton used for such an approach is called *observer* or *observing automaton*. If the automaton representing the model terminates with the observing automaton in an accepting state, the model satisfies the LTL-formula [6, 12].

In praxis the observing automaton is not constructed directly from the formula Φ, but from its negated form $\neg\Phi$ [12]. This process can be interpreted as searching for a counterexample. This notion has some advantages over the direct approach. The first advantage is, that if the observing automaton terminates, one immediately obtains a path that violates the properties. This path then can be used as a counterexample to fix the initial system or to correct the model specification. Secondly, the search for a counterexample has the advantage, that if the automaton violates the initial property, the exhaustive search of the negated property will usually more quickly find a counterexample, than it takes to search all the valid executions for a path that does not satisfying the property.

2.3.3 Symbolic model checking

An alternative to the verification techniques presented above is symbolic model checking. Instead of explicitly modeling the automaton and its states, this technique is primarily used for CTL, and tries to verify Φ by directly reasoning about Φ and the atomic prepositions at each state. For this, a set $Sat(\Phi)$[1] is constructed that stores the states which satisfy Φ. This requires the $Pre(\mathcal{S})$ operator, which stands for the immediate predecessors of statement $\mathcal{S}$ in context of the automaton. Before this method can be applied it must be possible to obtain a symbolic representation of $Sat(\mathcal{P})$ for every atomic proposition $\mathcal{P}$ of the automaton. Also a method to compute $Pre(\mathcal{S})$ for all symbolic representations is required. Further more one requires algorithms to compute the complement, intersections and union of these symbols. $Sat(\Phi)$ itself is defined by the Pre-operator, by $Sat(\Psi_i)$, sub formula Ψ_i of Φ, the set $\mathcal{Q}$ of states and the following rules[6]:

$$\begin{array}{lcl} Sat(\neg\Psi) & = & \mathcal{Q} \setminus Sat(\Psi) \\ Sat(\Psi \wedge \Psi') & = & Sat(\Psi) \cap Sat(\Psi') \\ Sat(EX\Psi) & = & Pre(Sat(\Psi)) \\ Sat(AX\Psi) & = & \mathcal{Q} \setminus Pre(\mathcal{Q} \setminus Sat(\Psi)) \\ Sat(EF\Psi) & = & Pre(Sat(\Psi)) \vee Pre(Sat(EF\Psi)) \end{array}$$

The advantage of this method is that the state explosion tends to occur less often and that one possibly can verify larger models with this method.

2.4 About models

So far it has been shown how models can be precisely specified together with their properties. At this point the question arises, if it is possible, to model any problem that one might want to study. Unfortunately, there is no straight answer. For some cases it might be possible to completely model the whole system and apply model checking, to discover flaws and to better understand the system, but for other cases this might not be possible.

There are several reasons for this. For a start, an exact representation might not be available to build a complete model of the problem at hand. For example for a network protocol one might not be able to completely describe all the possible error-sources for a network-cable, due to the sheer amount of different error sources. In an other example one might want to model a system of traffic lights, yet it is impossible to precisely model the behavior the all different traffic participants. In both cases one is forced to abstractly model these behaviors. Additionally, even for systems, which have been completely described, as with software, one still faces the problem that the

[1] Satisfies

full representation of the system would be simply too large for todays systems to store, nevertheless to verify. For this, one should remember, that even a system with four 16-bit integers variables has already 2^{64} possible states, which potentially would have to be searched during the verification phase.

Therefore one is almost always forced to create an abstract, less complex model of the system under study, in order to verify it. Thereby it is necessary to validate, if such an abstract system can verify the given properties. For this one has to show, that if the more simple abstract model $\mathcal{A}$ satisfies a specific temporal formula Φ, then the original system $\mathcal{O}$ also satisfies Φ.

To ensure, that an abstract model can be used to prove a CTL* formula for the original system, the executions of the original model must be a subset of the more abstract representation. In this case one also says, that the abstract model has to have more freedoms or behaviors than the original.

Applied to the sets of paths $\sigma_{\mathcal{O}}$ from the original system, this means, that every path has to be an element of the set of paths $\sigma_{\mathcal{A}}$ from the abstract model, $\sigma_{\mathcal{O}} \in \sigma_{\mathcal{A}}$. With this in mind, one then can use the notion of inverted temporal formula to search for a counterexample in the abstract model. If no counterexample can be found, the abstract system satisfies the temporal statement, but also the original-system ($\mathcal{A} \models \Phi \rightarrow \mathcal{O} \models \Phi$), because all the paths from $\mathcal{O}$ are a subset of $\mathcal{A}$. Unfortunately, if the temporal formula does not hold for the abstract system, there is a chance that the path that violates Φ in $\mathcal{A}$ might have been introduced by the abstraction and actually is not a path of $\mathcal{O}$. In the case that the path that violates Φ also violates the original model, then the violation is called *true error* or *true negative*. If the path violates $\mathcal{A}$ but does not violate $\mathcal{O}$, then it is called a *false error* or *false negative*. If all errors of a model are *true errors* the model is called *complete*.

However, which of both cases the path belongs to, can usually not be decided automatically and has to checked separately. Never the less, because the path that violates the property ϕ, called *error trail*, can automatically be constructed and a quick check against the original program or the model can often quickly decide if the path is indeed a valid error.

In general, there are two major methods to reduce the complexity of such a system. The first is reduction of data, be it reducing the size of variables or completely removing those variables, and the second is decreasing control information by merging states, removing states or restricting execution paths. It is also often important when choosing an abstraction to consider the properties one would like to verify, since this usually guides which portion of the original system can be safely abstracted.

An example for a data abstraction would be to replace any numeric variable with a 4-state variable, that states if the original variable was greater (+) or smaller (-) or equal (0) zero or of unknown value (?). This would reduce the number of possible executions for (Fig. 2.12) with $a, b \in [2^{16} \dots 2^{15}]$ from $1.8{*}10^{19}$ to 26 for $a, b \in [-, 0, +, ?]$. In this case, if either a, b or both are of unknown value the result of $a * b$ would be unknown and therefore all branches would have to be explored. This clearly follows the

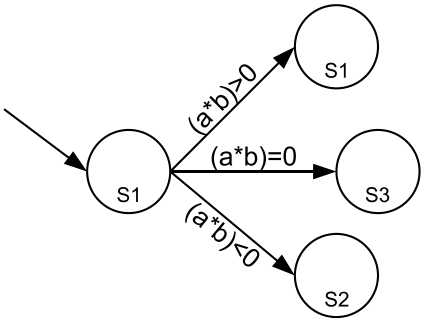

Figure 2.12: Example transitions for data abstraction example

notion that the abstract model has to contain all the paths from the original model.

The example should only be short introduction of how one could create an abstraction. There are many ways how one could create an abstract model and most of the design choices are specific for the problem to be abstracted.

2.5 Summary

This chapter introduced the notion of model checking and how the formal foundations of model checking can be defined. Usually the representation of the model is some form of finite automaton, modified to suit the model checking system at hand. The standard for expressing properties is temporal logic, which can argue about finite points of time and possible future behaviors. Furthermore extensions to model checking exist that enable for example the verification of systems with quantifiable time or real time systems. However the basics for verification are mostly the same for those systems and have therefore been omitted from this chapter and can be found in [6].

The process to verify a given model consists of three steps. In the first step a suitable formal representation of the problem must be determined and if necessary abstracted to reduce the complexity of the model. In the second step one has to specify the properties that the model should satisfy using temporal logic, which is usually done either in CTL or LTL. In the third step this automaton is either manually verified or an automatic verification process is applied.

The whole notion of model checking unfortunately is strongly limited by the complexity of the model and behaviors at hand, which often prevents most direct approaches of modeling and verification. Nevertheless model checking is a very powerful in determining safety and liveness properties.

3 Approach

This chapter will first give a small introduction to MPI, which is necessary for the remainder of this work. Subsequently the requirements for a MPI communication model will be highlighted. The last part of this chapter will introduce the selected model checking system and the tool used for the automatic model extraction.

3.1 Introduction to MPI

The wildly accepted MPI standard is often used to implement the communication of parallel programs.

Definition 10:
A program using exclusively MPI for its communication is called MPI-program *for this work.*

The functions specified by the interface provide the program with point-to-point communications, collective operations, group-communication and topology handling routines. For all of those routines MPI defines precisely the access for each parameter of any MPI-function as either being read-only, write-only or read and write, without specifying how this is implemented in a programming language. MPI also requires strict typing for its messages [22].

The communication functions of MPI are based on the principle of a point-to-point message-based communication between two processes with different synchronization and buffering modes. The functions MPI_Send and MPI_Recv are the most basic communication primitives in MPI with many variations that implement different behaviors, like unbuffered sends and receives. The complete signature of these calls can be found in listing (Lst. 3.1).

A complete MPI message consists of two parts. The first part describes the actual data being transmitted, consisting of the data itself, the number of data elements and the type of the transmitted data. For this several often used data types from the host language

```
MPI_Send(buf, count, type, dest, tag, communication group)
MPI_Recv(buf, count, type, source, tag, communication group, status)
```

Listing 3.1: Signature for MPI_Send and MPI_Recv

MPI-Type	C-type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
...	
MPI_FLOAT	float
MPI_DOUBLE	double
...	

Table 3.1: Selection of MPI types for C-programs

are predefined, but one can also define additional new data types. Table (Tbl. 3.1) lists a selection of the C-types supported in MPI.

The second part, called *message envelope*, consists of the source and destination identifiers, called *ranks* in MPI, a tag to distinguish messages and a communication group identifier. The communication in MPI is restricted to groups, which can either be the standard global group or a user defined group, which is subset of all available processes. The source and destination rank identify a sending and receiving processes in context of a such communication group.

A message can only be received by a process, if all the parameters passed to the receive-function match the data type and all the parameters of the envelope. Yet for some receive-functions the data counter may differ from the number of elements of the message. Additionally in point-to-point receive operations the tag and source parameter from the envelope can be wild carded to receive from any processes or any message with differing tag. All the parameters for a message, except the source rank for send operations and the destination rank for a receive operation, have to be passed directly to the MPI calls (Lst: 3.1).

The standard communication mode for many of the communication functions blocks execution until it can be guaranteed that the input parameters are free to be reused and the output parameters have been properly set. This provides considerable freedom to the different library for a efficient implementation. Each communication function usually has a non-blocking counterpart, that starts the construction or reception of a message and immediately returns control to the calling process. In this case the parameters passed to the function must not be accessed until MPI completes the asynchronous call, which can be queried by a separate function. The standard additionally defines a *buffered*, a *synchronous* and a *ready* communication mode for each function. These modes guarantee special completion condition for the calls, providing the means to precisely design a communication scheme. The buffered communication functions guarantee, that a MPI-call will use a process local buffer to ensure that a send operation does not block due to a missing receive. For the synchronous functions, MPI guarantees that the send operation

```
#include "mpi.h"
main(int argc, char **argv)
{
  MPI_Status status;
  int  processCount,
       rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &processCount);
  int array[10];
  if (rank == 0)
  {
    MPI_Send(array, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(array, 10, MPI_INT, 1, 0, MPI_COMM_WORLD,&status);
  }
  else //rank>0
  {
    MPI_Recv(array, 10, MPI_INT, 1, 0, MPI_COMM_WORLD,&status);
    MPI_Send(array, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

Listing 3.2: Example of a simple MPI-program

only returns once the matching receive-operation has started, thereby implementing a rendezvous operation. A ready mode send requires that the matching receive has already been posted before the sending operations is called.

Besides the basic send and receive operations MPI specifies functions for collective operations. A collective operation is a communication and computations event involving all members of a communication group. The simplest example of such an operation is the barrier, which blocks each process in a group, until all processes have called the barrier function. However, these collective operations do not add new functionality and represent often used communication patterns of send and receive.

The code example (Lst. 3.2) shows a simple MPI-program that exchanges the content of an array between two processes.

A correct MPI program must initialize the communication library with MPI_Init before any communication or other functionality of MPI is used. A program also must shut MPI down by calling MPI_Finalize before it terminates to be correct.

Tool	Model description	Specification
SMV	Network of automata communication with shared variable	CTL
SPIN	Communication automata	LTL
KRONOS	Timed automata	Timed CTL
UPPAAL	Linear hybrid automata	CTL
HYTECH	Linear hybrid automata	Control instruction set
Design/CPN	Colored Petri Nets	CTL
JPF	Communicating automata	LTL

Table 3.2: Selection of public domain model checkers [6, 26]

3.2 Model checking requirements

The most influencing decision when developing an automatic model extraction process for C-MPI programs is for which verification-system this model is going to be used for, since this directly impacts how and if at all the different programming-concepts of C and MPI can be represented in the model. At the time of this work, there were numerous model checking systems available, each with different capabilities and one has to choose, which of those systems supports the necessary feature or if alternatively a new model checking system supporting these features needs to be developed. A selection of different model checking systems can be seen in table (Tbl. 3.2), including their respective method of input and temporal specification. Since a full MPI program usually contains a large set of data and many control structures a direct model would be too complex for any model checking system to verify. Therefore one has to restrict the properties to those, that are the most prominent and important. With these important properties in mind, one then has to select a model checking system, that can express these properties and generate a suitable model from the program. For a PI application one might want to check data-properties, to verify if the computation is correct, or control properties to check if the program can crash or deadlock. Since any model checking system would be hard pressed to verify even the most simple data-properties due to the complexity of mathematical calculation and different input domains, it was decided to focus on control-properties for a MPI application. However, this does not directly imply that data properties can not be verified, just that the model extraction process will be focused on control-flow, thereby possibly abstracting data.

The control flow of an MPI-program is directly influenced by the control constructs of the C-programming languages like if and for. Additionally the control flow may also be blocked by MPI functions and influenced by data returned by the calls. Therefore, a model checking system must at least support the messaging operations capable of modeling the non-blocking MPI_Send and MPI_Recv functions and data types to enable message matching and basic control structures.

3.3 MPI-SPIN

For this work the MPI-SPIN [20, 21] model checking system was selected, which was designed to verify non-blocking MPI-programs. MPI-SPIN is an extension of the popular SPIN [12] model checking system and directly supports most of MPI communication functions. However, this model checker has some restrictions which for one part originate in the SPIN-model checking system and for the other part are based in the extension itself. For example, in the current version 0.1, MPI-SPIN can not model communication groups or buffered and ready communication modes. SPIN and therefore MPI-SPIN can not model floating point types, arbitrary arrays or dynamic memory. The extended model checker offers a symbolic alternative to floating point numbers by providing some symbolic numerical calculation functions and type as means for floating point abstraction, which is not used in the scope of this work. Nevertheless, this extended model checker supports all of the important concepts, like control-constructs, message-passing and variables, that are necessary to construct a meaningful model of the MPI-communication. Besides having the direct MPI model checking capability a second advantage in using a SPIN derived model checker is, that SPIN has some special constructs that promote automatic model-extraction processes. Therefore MPI-SPIN was selected for this work. A quick survey of other freely available model checking tools showed, that SPIN had in comparison no substantial shortcomings in terms of describable models or its ability for verification.

3.3.1 SPIN

SPIN is a model checker for LTL-formula with its own input language and is based on a direct verification using an exhaustive search of all possible executions to find a counterexample for the LTL-claims. This model checker is directly based on the idea of combined automata and verification by an observer presented in chapter 2. The model checker also supports search of deadlocks and lifelocks and can additionally check the model for user defined state properties (assertions). The SPIN model checker uses *PROMELA*[1] to describe the model, from which it constructs a verification automaton. PROMELA is somewhat a programming language with some similarities to C and supports deterministic and non-deterministic control constructs, several variable types and basic mathematical operators. Even user constructed types and small arrays of predetermined size are possible.

The sample code (Lst. 3.3) shows a simple model formulated in PROMELA and describes a system which randomly multiplies the variable i with 2, 3 or 5 until the value of i is greater 100. After doing so it prints *even* if i is even and *odd* if not and states that i is greater than 100. The active proctype[2] states that a process description is following, where the process is active from the beginning.

[1] PROMELA is an acronym for *process meta language*

[2] A proctype is a process definition in SPIN

```
1  active proctype example()
2  {
3    int i=1;
4    do
5    ::(i<100);
6      if
7      ::i=i*2
8      ::i=i*3
9      ::i=i*5
10     fi
11   ::else
12   od;
13   if
14   ::(i%2)==0;printf("even\n")
15   ::else;printf("odd")
16   fi
17   assert(i>100);
18 }
```

Listing 3.3: A sample PROMELA model

Every model described with PROMELA consists of at least one process, defined by init or proctype. These processes themselves are defined by a sequence of steps, that again are composed of statements or declarations (Fig. 3.1). The supported basic native variable types are the unsigned bit, boolean, byte, the signed or unsigned short and int. There are also PROMELA-specific types, like the chan-type representing message channels, the mtype for enumerations and some special types used for accessing the state-vector. Table 3.3 lists these types with their value range. Each of the types of (Tbl. 3.3) can be used to compose structures and one-dimensional arrays. Of course those elements can again be used for constructing more complex types based on previously defined types. Convectional comparators and mathematical operators can be used to manipulate or compare these variables.

In contrast to C, SPIN only has a two level scoping mechanism, from which follows that there may be only one unique variable in the global scope and only one variable of one name within a given process scope. Also there is some limited support for parameterized subroutines. Unfortunately these subroutines do not provide the means to return values and do not establish an additional scope, which prevents any direct form of recursion and local parameters. These subroutines are more like macros with parameters than actual functions and represent a convenient method of reusing often used code. In addition to user defined subroutines PROMELA provides some predefined functions which are the only routines that can return values. An example of such a function is the nempty(chan) function that returns true if the passed channel is not empty.

type	bits	signedness	value range
bit	1	unsigned	$0 \dots 1$
bool	1	unsigned	$0 \dots 1$
byte	8	unsigned	$0 \dots 255$
mtype	8	unsigned	$0 \dots 255$
short	16	signed	$-2^{15} \dots 2^{15} - 1$
int	32	signed	$-2^{31} \dots 2^{31} - 1$

Table 3.3: PROMELA types for 32 bit systems

The conditional statements in PROMELA are constructed by using IF and DO, which somewhat resemble the IF and WHILE constructs from C. The PROMELA IF selection construct marks different execution sequences from which only one is chosen and executed, with no predefined preferred selection of those sequences. Each control sequence might be executed if the first statement, called *guard*, is evaluated to true. In the case that there are multiple executable sequences one is randomly chosen without any preference. If no guard can evaluate to true, the process blocks unless the else guard is present. The use of guard and else can be seen in line 15 of listing (Lst. 3.3). With this description it is possible to define both non-deterministic as well as deterministic selection behaviors. Furthermore, any statement in PROMELA that has not a direct boolean association automatically evaluates to true. An application of this mechanism can be seen in lines 7, 8 and 9 in (Lst. 3.3). The DO construct behaves similar to the IF construct only that is does not exit after having executed one sequence but continues to choose sequences until the loop is exited with an explicit break statement or with a GOTO jump. In addition to those constructs, that are required for the construction of the model, PROMELA provides special constructs that support the verification process, like assertions and mechanisms to add temporal claims to the program. This was a small introduction to PROMELA, where many more constructs exist to precisely describe the concurrent model. These constructs as well as the one mentioned above are described in much more detail in [12].

However, the execution model and the model extraction support mechanism are worth further explanation. As already described in chapter 2 the model checking relies heavily on the notion of automata, with SPIN being no exception to that. The input-language PROMELA was specifically designed so that an automaton can be easily obtained from the specification. For this every parallel process described in the model is directly transformed into an automaton. Each sequence of statements within each process is represented by a sequence of states, where each transition describes a single PROMELA-statement (Fig. 3.3). Some constructs introduce branches to these sequences, where the IF and DO selectors are the most common ones used. These selectors precisely create as many different branches in the sequence as there are option sequences described (Fig. 3.4). A more elaborate example of such an automaton representation of

```
...
proctype: [active] PROCTYPE name '(' [decl_lst] ')'
          [priority] [enabler] '{' sequence '}'
...
sequence: step [';' step ]*
...
step    : decl_lst
        | stmnt [ UNLESS stmnt ]
...
decl_lst: one_decl [';' one_decl]*
...
stmnt   : IF options FI
        | DO options OD
        | ATOMIC '{' sequence '}'
...
        | ELSE
        | BREAK
        | GOTO name
        | name ':' stmnt
        | expr
        | c_code [ c_assert ]
        | c_expr [ c_assert ]
...
options : ':' ':' sequence [':' ':' sequence]*
...
```

Figure 3.1: Excerpt of the PROMELA-grammar in augmented Bachus-Naur form

a PROMELA-program is figure (Fig. 3.2), which shows a possible automaton for the model from (Lst. 3.4). In this example one can observe the mapping from PROMELA to an automaton. The final model is constructed by building the asynchronous product of all the automata described in the input, where each automaton is an instance of one of the processes. In the final automaton product each state usually has at least as many transitions as there were different processes in the model.

The verification is then performed by exhaustively exploring all possible executions of the verification automaton. To prevent a repeated analysis of previous analysed parts of the automaton a backtracking technique is applied during the exhaustive search.

Beginning from the start state each transition it is evaluated whether the guard is satisfied. If the transition can be performed, the state vector is saved and the transition is applied. From this point the process is repeated. In the case that the next state has already been previously visited, the new state-vector and the set of saved state-vectors for that particular state is compared. If a match of the current state-vector is

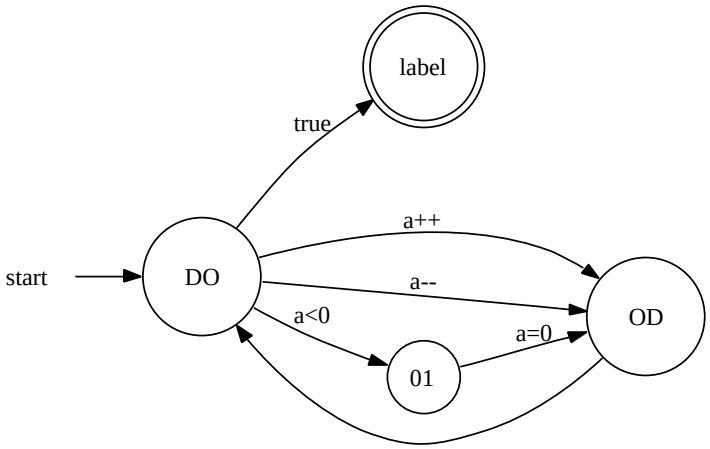

Figure 3.2: An automaton representing the model of (Lst. 3.4)

```
proctype example()
{
  int a=0;
  do
  ::(a<0);a=0
  ::a++
  ::a—
  ::true;goto label
  od
label: skip
}
```

Listing 3.4: A sample model

encountered the process backtracks. If not, then the state has not been visited with the current state-vector and this vector is again added to the set of state-vectors for the current state. After this the process proceeds again with exploring all the transitions.

This continued accumulation of different state-vectors for states is one of the main reasons, why the amount of variables and their range must be restricted since the number of elements in this set might become humongous, which SPIN tries to counter by applying different compression techniques [12].

This state exploration continues until an end-state is encountered. If the end-state is from the negated LTL automaton the verification process has found a counterexample. This proves the failure of the claim and the backtracking information is used to generate the error-trail. For any other end-state this means that the automaton of that end-state has terminated. If all non observer automata terminate, the process has found a

c_code {...}	direct specification of a C-code fragment
c_expr {...}	a side effect free expression that has to evaluate to true or false
c_decl {...}	declares type like structures and typedefs
c_state {...}	defines a variable in the state vector
c_track {...}	defines existing memory that is included in the state vector

Table 3.4: Model extraction support functions for SPIN 4.0 or higher

valid termination of the program and backtracks until another unexplored state-vector transition pair is found and proceeds from there. If no such pair can be found and no error-trail has been discovered, the model satisfies the LTL-claims according the underlying theory (Chap. 2.3).

The special case of this verification is the verification of deadlock freeness. For this the analysis verifies that for each non-end-state of the model at least one transition is always possible. If no such transition can be found then the model can deadlock. Further special case analysis techniques are available, but are not applied during standard verification [12].

The SPIN model checker uses a somewhat antiquated technique for performing the actual verification. Even though a simulation mode is available that is capable of interpreting any basic PROMELA model, SPIN does not use this interpretation mechanism for the verification. Instead SPIN acts as a verifier-factory for the model and generates a specialized C-source for a specific problem. This C-code itself is then compiled to produce the actual verifier. This methods is usually more conservative in terms of memory and performs better than an interpretation of the model. This also enables the model extraction support mechanism mentioned earlier.

The PROMELA support mechanisms from table (Tbl. 3.4) enable the specification of fragments of arbitrary C-code, that can be used to implement a transition or a guard. There are also mechanisms to add segments of memory to the state vector enabling the usage of a wide range of C-language features. Though C-fragments are very helpful when automatically extracting a model, these fragments pose a great risk to the validity of the verification and require a great deal of caution when used. The problem is, that it is possible to add state information to the automaton which the verification system is not aware of. Also it is possible to break the verification process by specifying non deterministic behavior within these constructs, which is a violation of the atomic transition rule for automata (Chap. 2.1). Therefore it is strongly encouraged not to use these mechanisms in human generated models. An automatic extraction process however usually can more precisely ensure that the used C-fragments do not include any unaccounted or non deterministic behavior to the model. This unaccounted behavior also includes most operating system interaction, which SPIN can not track. For example, if a mutex[1] object is used in such a fragment, the verification process of SPIN should not

[1]Mutual exclusion

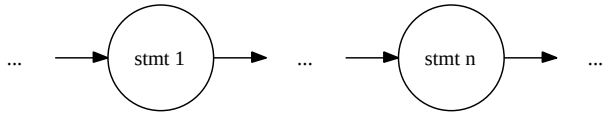

Figure 3.3: A sequence

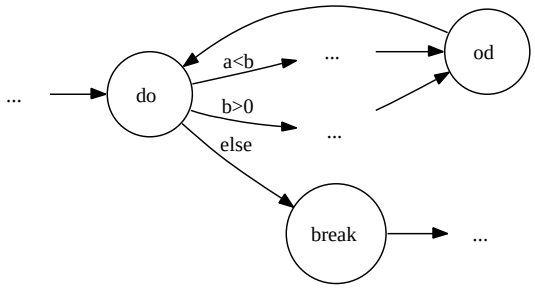

Figure 3.4: A DO statement

rollback to a state before that specific transition, without undoing the mutex operations. But since this additional state information is stored within the operating system and SPIN has no formalism to access and reset this information, the verification would produce undetermined results. This especially is true for most cases of IO[2]-operations, because with these operations it additional state information is created on the hard drive and in the operating system.

3.3.2 MPI-SPIN

As for the MPI calls one could utilize SPIN inherent mechanisms to model the communication between processes. The available message channels, which provide buffered or unbuffered communication, can easily be used to represent the blocking or buffered MPI send and receive operations. These channels also can not be used to describe the non-blocking calls and SPIN has no further concepts available to express this behavior [20, 21]. MPI-SPIN offers a solution to this problem by providing many of the communication mechanisms described in MPI directly within SPIN. On the other hand MPI-SPIN was not designed to a completely implement the MPI-standard for SPIN and has currently some restrictions on full support of MPI. This extension provides the MPI send and receive functions in their blocking and nonblocking variants, but not the buffered and ready versions. It also provides the basic MPI wait, test, probe and cancel operations and, as far as the data types allow it, even collective operations. Unfortunately it does not yet support the concept of group communication providing only global communication. Further restrictions exist to request handles, MPI error codes

[2]Input Output

and threading behavior, but his has more impact to the syntax within SPIN than to the modeling capabilities in regards to MPI.

MPI-SPIN also institutes a limit for buffered messages and for the number of outstanding requests. The limitation within SPIN serves the purposes to limit the increase of the complexity for the state-vector and the verification. A second advantage is, that even though no such formal limit exists for the original MPI specification, any system only has a finite amount of resources and no application using MPI should presume there is infinite memory for message buffering available. Therefore a program should still be valid if such a limit applies. Hence this limit serves also a useful purpose for the verification of a MPI-program.

Besides the already mentioned MPI functions, there are also new functions and variable types that are not specified in the MPI standard, which are useful when constructing a model. The new MPI type constants MPI_POINT and MPI_SYMBOLIC provide the means to generate MPI-specific abstractions for the data types used in send and receive operations. The MPI_POINT-type describes generic data with no size and is usually used in send and receive to indicate only the amount of elements transmitted. The second MPI_SYMBOLIC-type is used for the MPI-SPIN specific symbolic representation of a numeric values. The MPI_UNDEFINED is used to indicate uninitialized type-information. For the symbolic types a new set of functions is defined in MPI-SPIN that allows symbolic algebra [21].

Besides to the basic safety properties of assertions, deadlock freeness and the user defines properties, the MPI-SPIN model checker also verifies some basic safety properties of MPI. During the verification the extended model checker tests, if buffers of outstanding requests can intersect creating possibly invalid data, and that the limit of messages and requests is not exceeded. Other properties automatically checked are that all communication have to be finished at the point that MPI_Finalize is called and that any incoming message can not overflow the buffer it has been given.

3.4 ROSE

For an automatic model-extraction process for MPI-SPIN and PROMELA one needs to identify all calculations that somehow influence the message envelope of MPI and the send or receive count parameter of all MPI communication constructs. Additionally it is necessary to correctly implement all the control constructs that determine if a given MPI call is executed or not. An example for such a construct is an IF-statement, that has a MPI-call in its true-body. This IF directly controls if the MPI-call is executed. Therefore it has to be represented in the model. All other control-statements have to be checked for similar dependencies.

All this information has to be automatically extracted from the source code without user interaction. Text source code, as it is written by the programmer, is not very suitable for any analysis since it is very difficult and complex to work with. If one would

```
#include "stdio.h"
int main(int argc,char ** argv)
{
  int a=1;
  printf("Hello World\n");
  return a+1;
}
```

Listing 3.5: Sample "hello world" program

write direct analysis for source code, it is necessary to constantly use string operations to identify the various variables, C-identifiers and comments. In such a case, it is also necessary to implement some form of error checking to prevent a faulty model extraction for invalid C-programs.

For theses reasons usually a more abstract representation of a program is used. This representation is called *abstract syntax tree*, usually abbreviated with AST, and originates in the domain of compiler technology. The AST does not have a single formal definition since it is more a concept than a defined construct and is usually adapted for a specific problem. For program source files the AST is usually a tree like representation that precisely describes the structure of the program without having to rely on the literal form of the source constructs. Often the native literals from the programming language are represented as specialized nodes. Functions and variable names are replaced by identifiers removing the need for repeated string comparisons. Also variables are often stored in a resolved fashion, such that it is possible to quickly determine where in the code a variable was defined and of what type the variable is of. In such an AST the internal nodes usually represent operators and the leaves represent the operands of the operators. Usually the parent-child relation in the AST states that the child nodes are somehow nested within the parent node in the source program. For example, this can be a statement within a function, where the function is a parent of the statement, or the parameter to a function call, where the function calls is the parent of the parameter.

Figure (Fig. 3.5) shows a simplified AST-representation of the program (Fig. 3.5). Due to these AST representations it is much easier to write analyses and transformations of a program. The construction of such an AST, called *parsing*, for the C-programming language is a complex and not very trivial task. Therefore it is not advisable to newly develop a parser but to use an available tool. Most common used compilers however tend to generate ASTs that have been optimized for the specific code generation process and often discard some information during the parsing process. Since the model should at least have some similarities to the original source these compilers are not very well suited for this work.

The ROSE-Project [17, 19] is a source-to-source compiler framework specifically designed to provide an easy platform to gain access to an AST for optimizations and

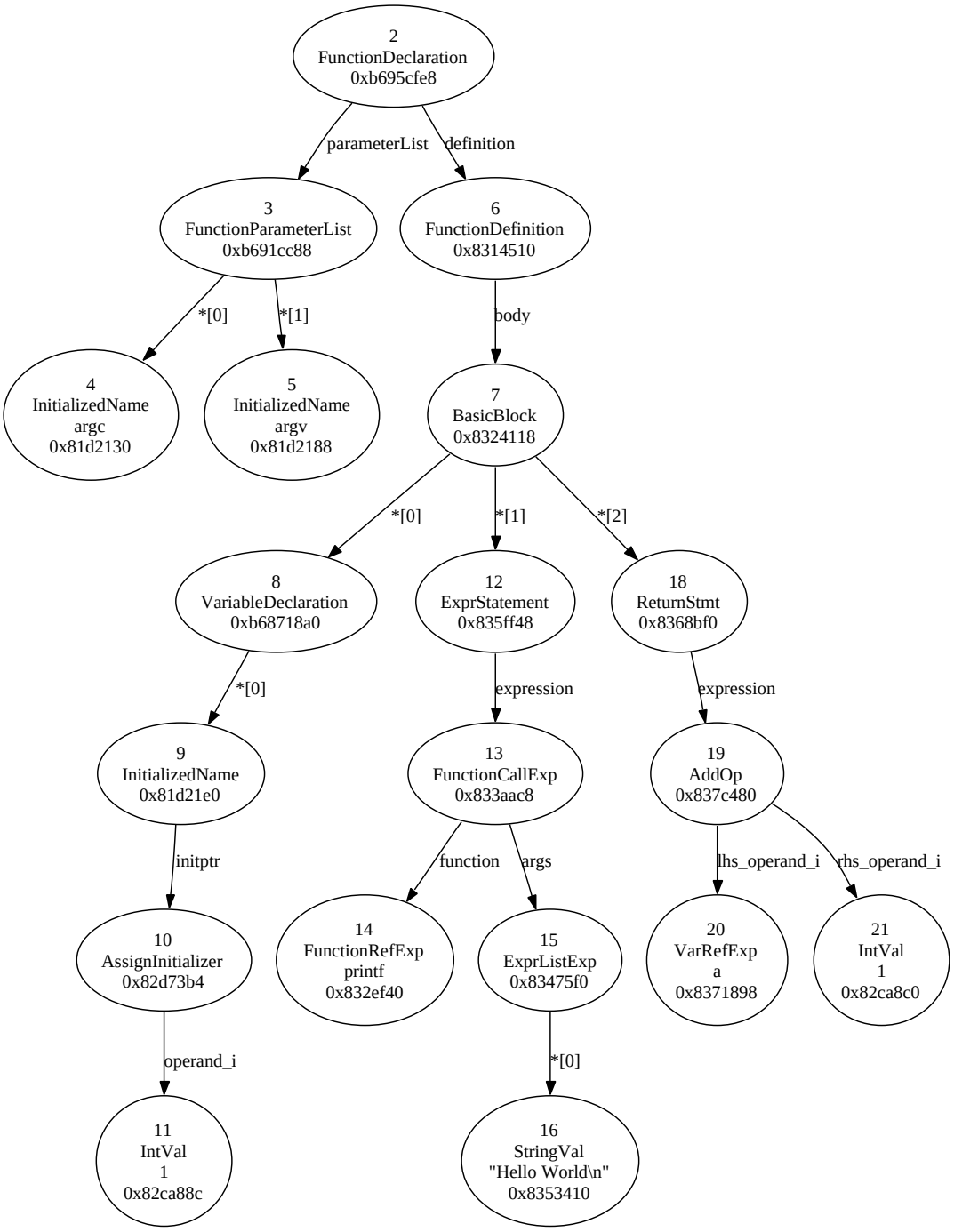

Figure 3.5: Simplified ROSE-AST of listing (Lst.3.5)

transformations. In this case a source-to-source compiler is a transformation tool that parses the source files of a program, constructs the AST and generates again source files from the AST instead of generating an executable code.

The process within ROSE is split up into the following three phases. In the first phase, which is not designed for user interaction, the framework reads a C-program and constructs a very powerful and rich AST, which contains all information necessary to reconstruct the original program. In the second phase the user of the frame-work can now specify transformations, apply existing or new implemented analyses and alter almost all of the aspects of a program. In the third phase a now source file is composed from the AST. An advantage with ROSE is, that it has several useful and commonly used analysis tools and transformations available, like control-flow graphs, type-informations, call-graphs and search functions. Some selected available transformations are function-inlining and loop-transformations like loop-fusion, loop-unrolling and for-to-while-transformation. Additionally ROSE provides additional functionality for directly modifying the AST-nodes, like delete, move and copy functions. Of course the user can implement new transformations, node-construction and additional analyses.

3.5 Summary

The first part of this chapter provided some background of MPI and highlighted the parts of MPI that are necessary for a model checking process. For any meaningful MPI model that is used to verify the control-aspects of the communication structure all calculations that affect the envelope and the amount of data being communicated needs to be contained in the model. In the second part the tools and verification systems used for this work were introduced. Is was explained, why MPI-SPIN, an extended version of the popular SPIN model checker and its input language PROMELA are a suitable tool for the verification of MPI-C-programs. The ROSE framework is going to be used as a base for the analysis and transformation of the MPI-program to PROMELA.

With the knowledge from chapter 2 and the tools introduced here is should be possible to automatically extract the MPI-model for MPI-SPIN for verification. The general idea of the model extraction is to analyse the program for its control and data properties, determine which of those elements affect the MPI-interface and extract and transform those elements to a model using either elements of PROMELA or by encapsulating portion of the program in C-blocks in PROMELA. Chapter 4 will explain, how this can be done in more detail.

4 Model extraction

This chapter covers the implementation and the model extraction process in some detail. The fundamental and important steps involved in this extraction process will be described as well. However, some of these steps are based on techniques from compiler technology and are not explained in detail. Information on general compiler technology and its methodology can be obtained in [1], in [7] or any related literature.

The process of the model extraction also makes heavy use of the AST-representation of ROSE and many of its mechanisms, which are to vast and complex to be describe in this work. Though a quick introduction has been given in chapter 3. Should these analyses and mechanisms deviate from the common standard, this will be especially noted.

As already stated, the general idea is to first identify all MPI relevant statements, calls and variables used in the original source program. In a second step the program elements that have an impact on the envelope of a MPI message are computed. With this information then required elements for the communication model are computed and the PROMELA representation is constructed. Since all the types of MPI's control parameters, mostly integer values or type identifiers, are supported in SPIN, no abstraction of such a parameter should be necessary. However, if the C-program contains non MPI-library functions or system calls which the developed process can not abstract, an automatic model construction is not possible. To ease development of this process, it was assumed that the source program does not contain aliasing. The reason for this is described later as well.

The process in general consists of three phases, an identification, an abstraction and a transformation phase, which will now be described.

4.1 Identification

In the identification phase the AST of the program is traversed and the MPI constructs used in the source program are identified. In this work, as already stated, the verification of the MPI-communication is the primary goal. Therefore these constructs solely consist of all MPI-functions used, such as MPI_Isend, MPI_Irecv, and their direct support routines like MPI_Init, MPI_Rank, MPI_Wait and MPI_Status. These calls are marked for later processing using ROSEs AST-attribution mechanisms with a MODEL_TARGET marker. Furthermore these calls, referred by the corresponding AST-nodes, may not be abstracted during a later phase and must exist in the later model.

4.2 Abstraction

The second phase tries to reduce the complexity of the model. In this phase the transformation also attempts to abstract any non-supported constructs, like floating-point types, large arrays and dynamic memory, to be able to construct a model. If unsupported constructs remain after this phase, the extraction process can not be continued. For the reduction of complexity and abstraction the established software abstraction technique called *static slicing* [8] is very useful. Since this work bases heavily on this technology, slicing[1] will be explained in detail.

4.2.1 Idea of slicing

Static slicing is a method to identify a set of elements in a program that have a dependency to a specified element, called slice-target, with two possible different variants of dependence. If one computes the set of program constructs such that every element in that set has some influence on the slice-target element, this method is called *backward slicing*. This backward slicing is precisely what is required for this work, because it computes for one selected MPI-call all the statements that the model has to contain. The other slicing method computes the opposite to the backward slice. By computing the set of elements that are influenced by the slice-target one computes the so called *forward slicing*.

With either set one can now reduce the program by removing any construct not listed in the set. This reduced program however is not guaranteed to be free of syntactic errors and often will not compile without further processing. To counter this one has to post-process such a slice to obtain an executable sliced program, also called *executable slice* [7]. The slicing technique used in this work provides such post-processing and therefore generates a executable slice. Either slicing method can be applied with different levels of detail. One could for example perform slicing for statements and expressions or for operators and operants, granting different detailed levels of insight into the structure of the program. For this work the main interest is on statements and expressions since the detailed dependencies for subexpressions do not directly influence MPI-calls. The constructs that the slice is computed for, called interesting nodes in this, are primarily statements and conditional expressions. A conditional expression is any expression used as the condition in a branching construct, like IF or FOR. Additionally, function-call expressions are also important, not only because they might be MPI-calls. A function-call is important, because a function itself is again composed of statements and one has to follow any statement for correct slicing. Therefore it is necessary to resolve function-calls during the computation of the slice by adding function-calls and their parameters to the set of interesting nodes (Def. 11).

[1] In this work the term static slicing is often used, therefore the prefix of static is sometimes dropped and slicing and static slicing will be interchangeably used.

Definition 11:
The interesting nodes of the AST for this work are:

- *Function-call expressions*
- *Statements*
- *Conditional expressions*
- *Expressions whose parent is the argument list of a function-call expression*
- *Variable declarations whose parent is the function definition*

In order to implement a slicing algorithm it is possible to directly operate on the AST without any intermediate representation by traversing and marking nodes to construct the final slice. This method however is complex, difficult to implement and to maintain, because one would have to handle every program statement separately and also would have to repeatedly track changes to the code. An alternative better structured method first constructs a dependence graph, used in vectorizing and parallelizing, and uses this graph to compute the set of statements in a slice [8, 16]. A dependence graph is a multi graph describing the dependencies between nodes with directed edges. Each of the edges is annotated with the type of the dependence between elements and often used dependencies are the data and the control dependency. There are several definitions of dependence graphs[16, 7] whereas the concept for all definitions is basically the same.

With the use of such a representation the process to obtain a slice turns into a simple graph reachability problem. To compute a backward slice with this graph one identifies the interesting node corresponding to the slice target and traverses recursively all incoming edges backwards, adding the visited nodes to the slice set. Once all possible nodes have been visited the set of interesting nodes represents the backward slice. The forward slice can be obtained in a similar fashion by simple recursively following the edges in their direction.

Before the algorithm is presented it is necessary to describe the dependence graph and its construction in more detail.

4.2.2 Dependence graphs

A dependence graph consists of nodes representing elements of a program, like statements and expressions or operands and operators, and directed edges. In such a graph an edge $A \overset{dependencetype}{\rightarrow} B$ denotes that for the correct operation of construct B the construct A, or some part of A has to be previously executed. In a dependence graph also multiple edges between two nodes are allowed, each indicating a different type of dependence, like the often used control and data dependencies. The type of the edges is not restricted to these types and custom edges, like call-edges that connect a function call with the function entry, are also often used to describe more detailed dependencies in a program.

Due to the flexibility of this concept different types of dependence graphs were defined, each describing a different set of dependencies at a different level of a program. The most prominent dependence graphs are the *system dependence graph* and the subsets there of, namely the *program dependence graph*, the *control dependence graph* and the *data dependence graph*.

A data dependency graph, short *DDG*, describes, as the name already implies, the data relationship between nodes. The control dependency graph, called *CDG*, explicitly shows the control dependencies between interesting nodes. The program dependency graph, *PDG*, describes control and data dependencies in one graph, but does not show, as the name implies, the dependencies for a whole program [9] since the graph can not describe concept of function-calls. A PDG describes only one function at a time and is a historical defined term.

This interprocedural information is included in the system dependence graph, called *SDG*, which is a total description of a whole program [7]. A SDG can be seen as a set of different initial independent CDGs and DDGs describing each single function and additional information describing function-calls, interprocedural data- and control dependencies.

Because the sketched slicing algorithm requires a SDG to correctly compute a slice of the program and ROSE did not provide any dependence graphs, a SDG analysis has been implemented for this work. A typical approach for the computation of a system dependence graph is to separately generate the control dependency and the data dependency graphs for each function. These individual graphs are then merged to construct a preliminary version of the SDG. This preliminary version is then completed by connecting each call site with the function being called. A call site is the expression where a function is called. In addition to the basic functionality the SDG description in this work has been extended with additional edges and nodes to ease the syntactic correct computation of an executable slice.

To simplify the explanation some nodes and edges required later for the construction and composition of the SDG are going to be explained during the construction of either the DDG or the CDG, depending to which dependency the belong to.

For this work the nodes of dependence graphs represent interesting nodes (Def. 11) by using a unique identifier to refer to the corresponding interesting AST-node.

Data dependence graph

The data dependency graph represents data dependency between variable assignments and the use of these variables, very similar to a *def-use chain* [15], abbreviated *DU-chain*.

The *def-use analysis* computes for a single definition of a variable, i.e. an assignment to variable, all the uses of the variable after the definition without crossing another definition of the same variable, i.e the variable maintains the value assigned in the definition. The counterpart of a def-use analysis is the *use-def analysis*, that computes for a use of a variable the definition that gives the variable its value. Though the

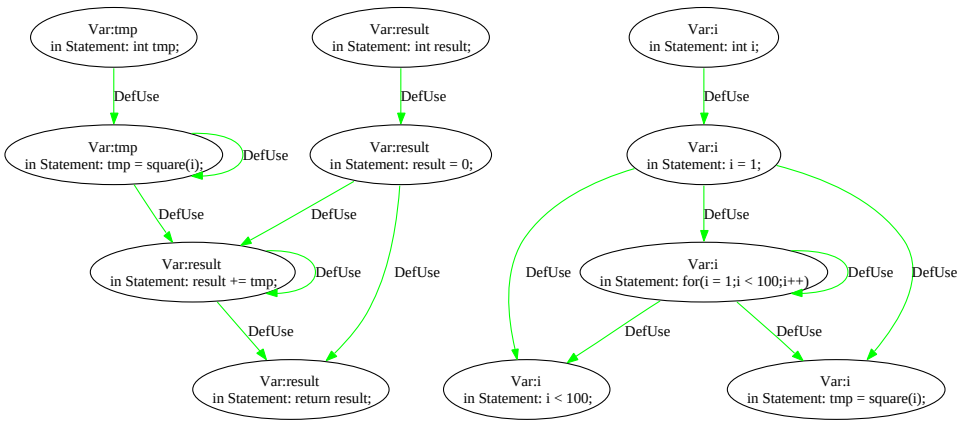

Figure 4.1: Def-Use chain for the example program (Lst. 4.1)

computation of a precise UD-chain is impossible, because data flow always depends on the actual control-flow during an execution [7], it is possible to construct a coarser chain that discovers the set of all possible definitions for a single use. Which of those definitions actually defines the variable can however not be stated. By connecting all the separate DU or UD relations the *def-use-chain* is obtained[15]. Figure (Fig. 4.1) shows such a def-use chain for the example program from listing (Lst. 4.1) and the definitions and uses for the variables are clearly recognizable.

The difference between a DU-chain and the data dependency graph is, that the DDG does not describe the dependencies for a single variable, but describes the dependencies for the statements and expressions in which the variable is used. A data dependency graph therefore merges the def-use relationship of all variables in a statement and therefore describes the data requirements for that particular statement.

This simple approach however is not capable to describe the more complex data dependencies that might occur for variables used in function parameter expressions, since C-programming language allows passing of pointers of variables to functions, thereby creating hidden definitions. A conventional intra-procedural def-use analysis can not discover such a definition. To circumvent this problem any variable used as an argument to a function is assumed to be newly defined by that function after the call. For later use in the system dependency graph the interface to functions is split into different sets of nodes. For this the parameters passed to a function-call are separated in input parameters, called ACTUALIN, and output parameters, called ACTUALOUT. Even though the parameter identified by an ACTUALIN and ACTUALOUT pair is the same, the data dependency changes between the call. Before the call, the ACTUALIN represents the parameter being passed to the function without a definition. The corresponding ACTUALOUT represents the variable, after possibly being newly defined in the function. In addition an ACTUALRETURN is added to represent the return value of the called function and is identified by the function-call node itself. This is a small extension for

```
int square(int x)
{
  return x*x;
}
int main()
{
  int i;
  int tmp;
  int result;
  result=0;
  for (i=1;i<100;i++)
  {
    tmp=square(i);
    result+=tmp;
  }
  return result;
}
```

Listing 4.1: Example program for data dependence analysis

the standard representation of function-calls [7], which is only defined for procedures without a return value. The use of ACTUALRETURN enables the representation of functions returning values for dependency graphs. Additionally FORMAL nodes are used to represent the interaction of the function itself with its calling parent, where the FORMALIN parameters are used to represent the input to the function and the FORMALOUT and FORMALRETURN are used to track outgoing data-dependencies. Again the FORMALRETURN is an extension of the notion of FORMALOUT nodes.

These ACTUALIN, ACTUALOUT and ACTUALRETURN nodes and their FORMAL counterparts together with data dependencies have to be correctly accounted in a data dependency graph, if the graph is going to be used for the construction of a system dependency graph.

Additionally the concept of pointers forms another difficulty when computing a data dependence graph. A pointer potentially can refer to any memory location used in the program, which can be location of a variable or some memory that was allocated using system calls like malloc or new. If a pointer variable refers to a memory location that another variable or memory location points to this is called an *alias*.

Typically every use of a pointer variable on the left hand side of an assignment is defined as a definition for the def-use analysis. This is also the case, if a pointer variable is merely used as the base address for an array operation. This possibly leads to very long def-use chains, since every use of a pointer variable on the left hand side is assumed to be a definition without the actual value of the variable chaining. Such a def-use chain only has limited detail of the actual data dependencies and would be hampering the

effectiveness of the slicing.

One way to counter this is to use a alias analysis like the points-to-algorithm [23] to more precisely determine which memory location a pointer refers to and to use that information to compute a more precise def-use information. Since there was no alias analysis available in ROSE at the time of this work, this could not be used.

Definition 12:
The function INTERESTING(x), where x is a node from the AST, computes the interesting node of x in the AST by ascending upwards until a interesting node is encountered. If x itself is a interesting node, x is returned.

The following pseudo-code (Lst. 4.2) describes the algorithm that was implemented to compute the data dependency graph:

```
RefSet= get variable references of function F
for varRef in RefSet do
{
  // get the target-node for the dependence
  Dest = INTERESTING(varRef)
  if (Dest == Function-call Parameter)
    RefDDNode = get ACTUALIN node for Dest
  else
    RefDDNode = get normal node for Dest
  // get all definitions for the use
  DefSet = DUChain.get definitions(varRef);
  for VarDef in DefSet do
  {
    Def= INTERESTING(VarDef)
    if (Def == Function-call Parameter)
      DefDDNode = get ACTUALOUT for Def
    else if (Def == Function Argument )
      DefDDNode = get FORMALIN for Def
    else
      DefDDNode = get normal dependence node
    establish data-dependence edge from DefDDNode to RefDDNode
    // get all uses to check for pointers
    UseSet = DUChain.get uses (Def)
    for PointerUse in UseSet do
    {
      if (left hand side(PointerUse) && !DUChain.is Def(PointerUse))
      { // a write to a pointer
        Use = INTERESTING(PointerUse)
        if (Use == Function-call Parameter)
          UseDDNode = get ACTUALOUT for Use
        else
```

```
          UseDDNode = get normal dependence node
        establish data-dependence edge from UseDDNode to RefDDNode
      }
    }
  }
}
ReturnSet = get return statements of F
FormalReturnDDNode = get FORMALRETURN of F
for return in ReturnSet do
{
  ReturnDDNode = get normal dependence node
  establish data-dependence edge from ReturnDDNode to
                                      FormalReturnDDNode
}
// link the return node to the parent statements
FunctionCallSet = get function-calls in F
for call in FunctionCallSet do
{
  ActualReturnDDNode = get ACTUALRETURN for call
  ParentInteresting = INTERESTING(call . get parent)

}
```

Listing 4.2: Pseudo-code for date dependence graph construction

Figure (Fig. 4.2) shows the data dependence graph computed for the sample program from listing (Ref. 4.1).

Control dependence graph

Like the data dependency graph, a control dependency graph describes the dependencies between interesting program elements in reference to their executability. A node A is control dependent on B, if there is a directed edge from $B \overset{control}{\rightarrow} A$ in the graph. Control dependence has here the meaning, that the element B in some form controls if A can be executed or not. For example if B is an control expression like the i<100 in the for-construct for(int i=0; i<100; i++), every statement in the body of the loop is control dependent on the expression i<100. This also applies to multiple parent constructs, like in the case that the for-loop is nested in another control structure. A control dependence graph however only lists the dependencies that directly control a node.

The control dependence graph can be constructed with different methods, like with a direct analysis [5] that works well with most programs. A more general approach [16] uses the control-flow graph and post-dominators to construct the control dependency graph for arbitrary programs. Since there is already a robust control-flow graph available in ROSE it was more feasible to implement the more generic approach [16] which required

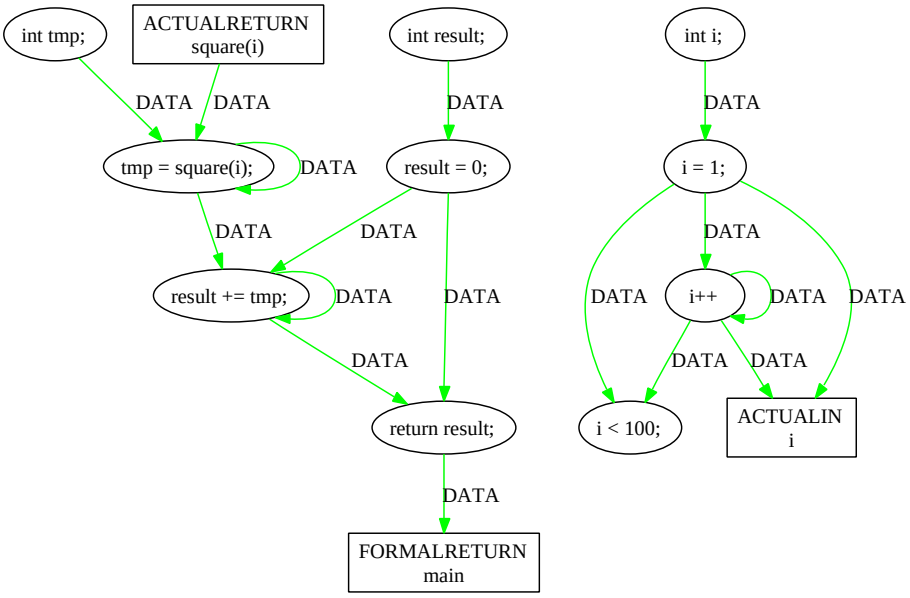

Figure 4.2: Data dependence graph for program of listing (Lst. 4.1)

the implementation of a dominator analysis.

A control-flow graph [16], abbreviated with CFG, is a directed graph of nodes and edges representing the transfer of control between parts of a program. Typically one constructs a CFG for a single function and computes a separate *call-graph* to represent control-flow between functions. Each edge has either a true or false attribute, expressing whether the source node to evaluate to true or false for that edge to be active. In addition to the existing elements from the function, the control-flow graph has two special START and END nodes. These nodes represent the entry and exit of the current function. To ensure that the start node is sllways the root of the control flow graph and that the exit node is reachable an additional false edge is established from the START to the END node. This false-edge from START to END, represents that the function is not called and is required by many analyses. The edge from the start to the first statement of the function is labeled true. For deterministic programming languages , like C, every node in the control flow graph has either two outgoing edges, a true and a false edge for conditionals, or just one true edge, representing unconditional transition of control flow. Figure (Fig. 4.3) shows such a control flow graph for the example program (Lst. 4.3).

A *dominator* in computer science is a special relation between two nodes of the control flow graph. A node X from the control flow graph G, dominates a node Y from G if all directed paths from the START node to Y have to pass through node X. The *post dominator* relation is a similar construct with the difference, that one does not examine the path from the START to a node, but from the node to the END. The CFG node X is post-dominated by a node Y, if all paths from X to END contain at some point Y. In principle the post dominance is a dominance on a reversed control flow graph.

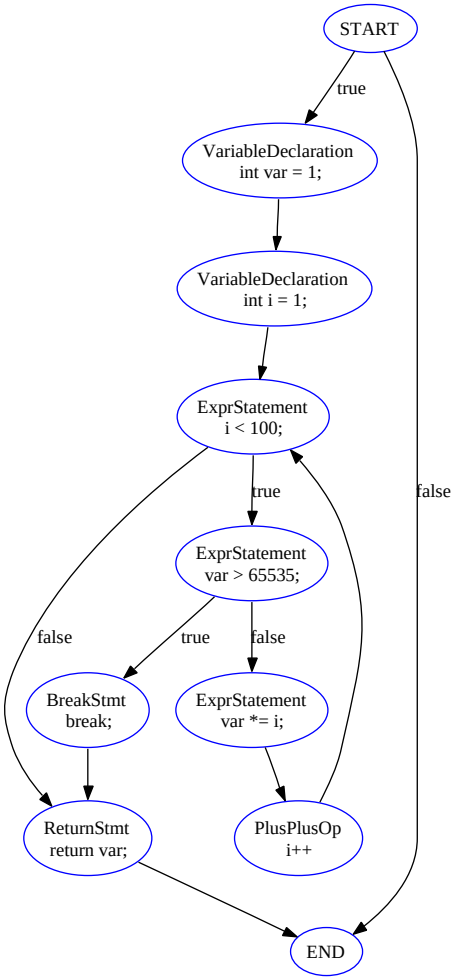

Figure 4.3: Control flow graph for the example program (Lst. 4.3)

Per definition a node dominates itself and as a logical conclusion a node can not post dominate itself. For any node N one also specifies a immediate dominator, called idom(N), as the node that dominates N but does not dominate any other node that dominates N without being N itself [13]. With the same definition one also obtains the immediate post dominator postIDom(N). With the control flow graph and the post-dominance the control dependency is defined in [16] as follows:

Definition 13:
Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

1. *there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and*
2. *X is not post-dominated by Y.*

```
int main()
{
  int var=1;
  for(int i=1;i<100;i++)
  {
    if (var>65535)
    {
      break;;
    }
    var*=i;
  }
  return var;
}
```

Listing 4.3: Example program for control dependency analysis

This definition of control dependence is rather complex at the first glance. The second rule, is the main indicator for a control dependence. If a node Y does not post-dominate X, this means there is one path that does not contain X which leads to Y, therefore Y can not control depend on X. On the other hand if Y post-dominates X, then per definition all paths to Y must pass through X. Therefore X controls Y provided that there is a path to Y in the control-glow graph. As an implication to this statement X must have two edges in the control flow graph. The first rule ensures that during the construction of the control dependence only the closest controlling node X is selected.

For the implementation of that algorithm to compute control dependencies a dominance analysis had to be implemented. The computation of dominators has been a well covered topic and several algorithms have been posted over time. For this work a modified version of the original control-dependence algorithm developed by [13] was implemented.

This original algorithm works well for all the control structures in a program. However, it fails to address, that statements like return, break and goto have a direct influence on the control-flow graph and are necessary for control-dependencies, which the given analysis fails to recognize. These nodes would without further treatment be represented as leafes in the CDG, though without them the control-flow graph would be different. Therefore they need to be represented in the CFG. This can be solved with a simple workaround. If during the traversal from the post-dominating node to the dominated node such a statement is encountered, an additional helper-edge from the statement to the controlling node is inserted. With this additional edge the control dependencies are correctly represent. The modified algorithm can be seen in the following pseudo-code of (Lst. 4.4). Figure (Fig. 4.4) shows the control dependence graph for the program (Lst. 4.3).

```
for all edges X,Y from the control flow graph
{
  if ! B postDominates A
  {
    tmp = B
    ADepNode = get normal dependence node A
    while (tmp != postIDom(A))
    {
      TmpDepNode = get normal dependence node tmp
      establish control-dependence edge from ADepNode to TmpDepNode
      if (TempDepNode == goto || TempDepNode == break ||
          TempDepNode == return || TempDepNode == continue)
      {
        establish helper-dependence edge from TmpDepNode to ADepNode
      }
      tmp=postIDom(tmp)
    }
  }
}
```

Listing 4.4: Pseudo-code for control dependence graph construction

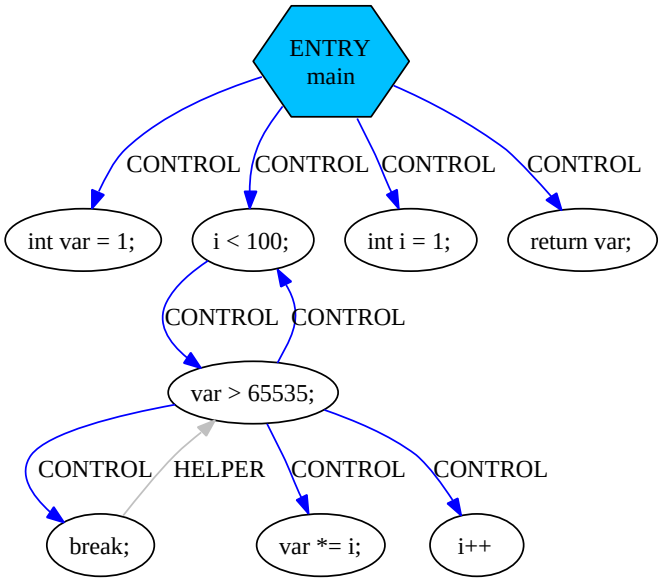

Figure 4.4: Control dependence graph for the example program (Lst. 4.3)

System dependence graph

Now the system dependence graph can be constructed by combining the control dependency and data dependency graphs for every function in the program and bay adding additional edges, representing interprocedural data and control flow. The original approach to the method described here was proposed in [7] and was slightly modified to better suit the needs of this work.

The general idea is to compute a program dependency graph for each function, which is the union of the data dependency graph and control dependency graph of that function with additional ACTUAL and FORMAL nodes, which have already been described along with the data dependence graph.

In the final step these PDGs are added to the system dependency graph and the interprocedural connections are established with PARAMETER_IN, PARAMETER_OUT and CALL edges.

For each function-call, the actual call site is connected to the function entry of the called function with a CALL edge as well as every ACTUALIN parameter is connected with the corresponding FORMALIN using PARAMETER_IN edges. Furthermore every FORMALOUT is further more connected with the ACTUALOUT using a PARAMETER_OUT edge. An augmentation often used for slicing, extends the SDG with additional summary edges between ACTUALIN and ACTUALOUT nodes to describe the internal data flow for the called function.

The difficult part during the construction of the SDG is the computation of these summary edges. Several methods for the computation of these edges exists [18, 7] and for this work the algorithm from [18] was implemented due to its improved performance.

Since all the PDGs essentially are subgraphs of a SDG there is also no explicit process to merge these PDG together to form the SDG. Instead the individual control dependency graphs and data dependence graphs are directly added to the system dependence graph without the intermediate PDG construction. To compute the summary edges a global analysis of the whole SDG is necessary. The full algorithm is described in much detail in [7].

The idea of this algorithm is to process the whole program simultaneously by back-tracking from every FORMALOUT edge of every function. If the back-tracking process reaches a FORMALIN node a summary-edge has been computed and is inserted at all call sites. To ensure correctness, once such a edge is inserted, the function has to be analysed again. Since this algorithm is complex and no modifications were made to it, one may refer to [7].

The final SDG for both sample programs (Lst. 4.1) and (Lst. 4.3) can be seen in figures (Fig. 4.6) and (Fig. 4.5).

So far it has not been shown how to represent dependencies for global variables in such a dependence-graph. The original solution to extend every function of the program with every global variable [7] was rather unsuitable, because such an augmentation of functions calls would seriously disrupt the mapping from MPI-calls in the source

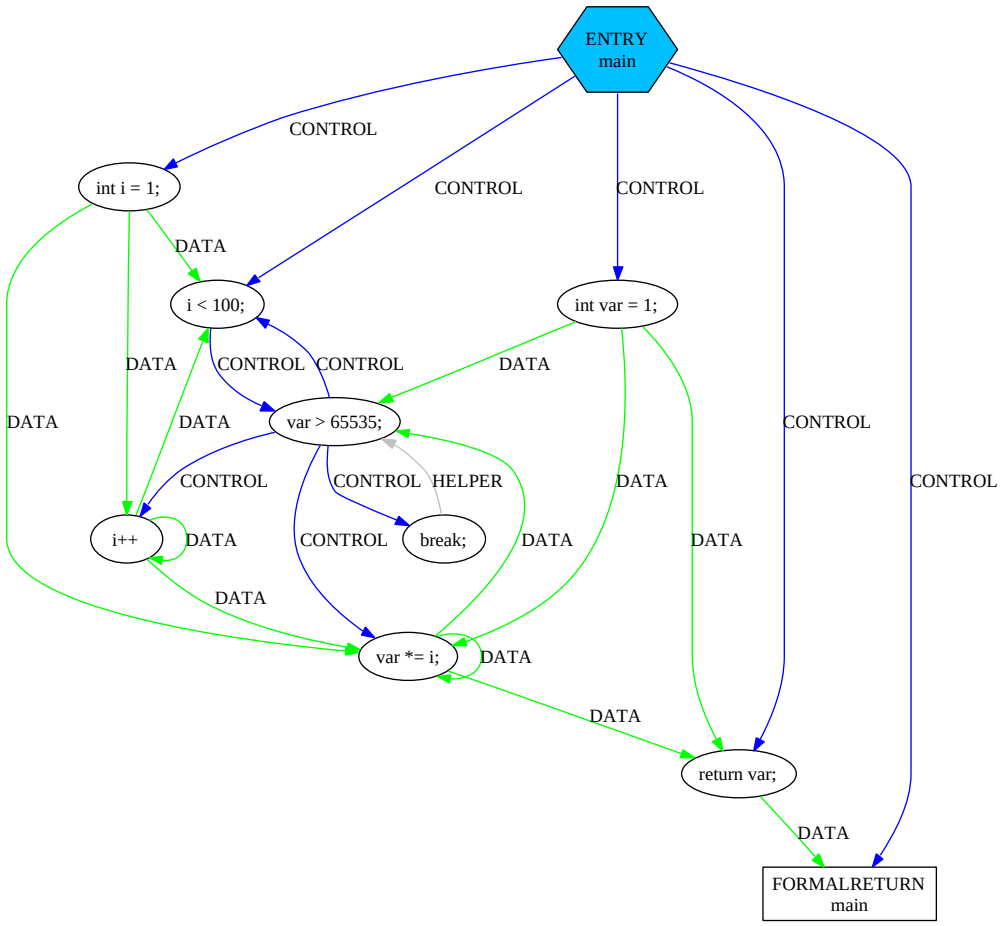

Figure 4.5: System dependence graph for the example program from the control dependence graph (Lst. 4.3)

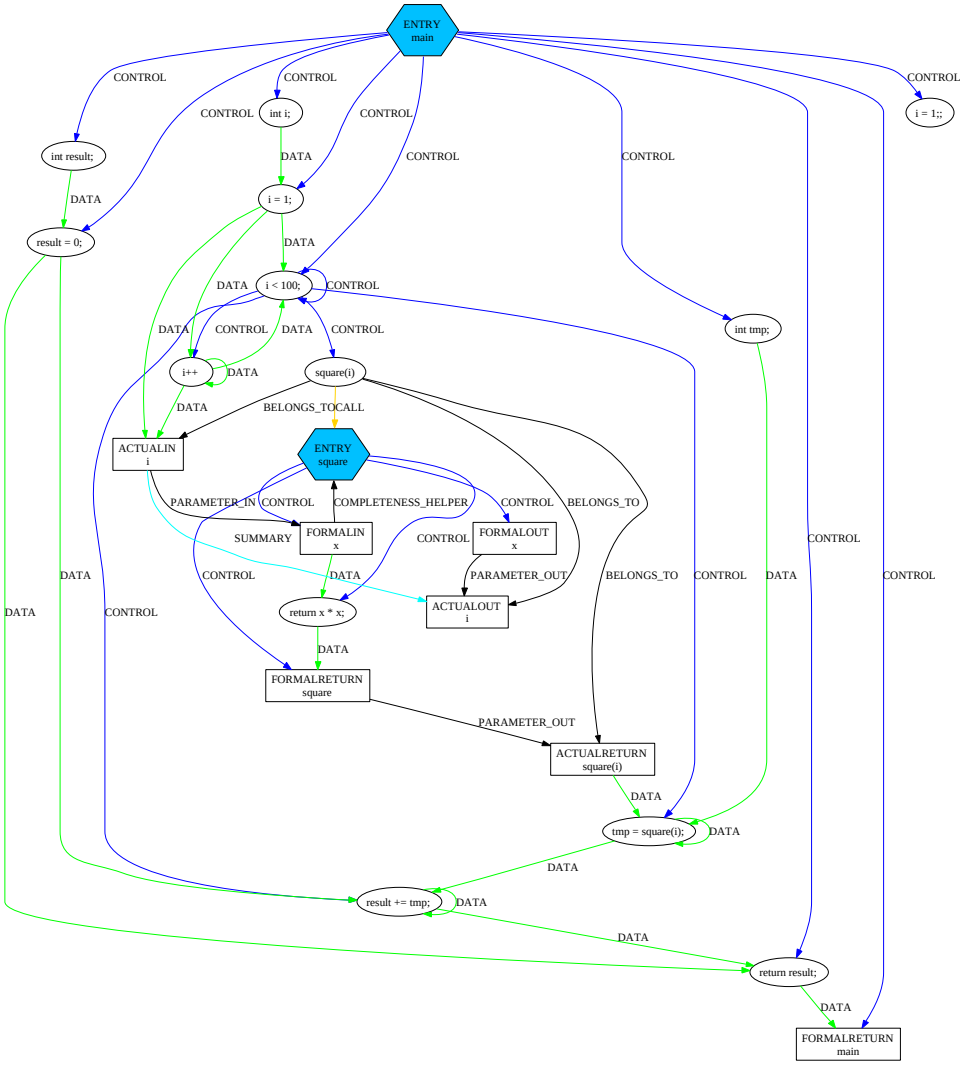

Figure 4.6: System dependence graph for the example program from the data dependence graph (Lst. 4.1)

program to their MPI-SPIN counterparts. This can be resolved by assuming that every assignment to a global variable creates a new definition without overwriting the old one. This causes every global variable to data depend on all definitions of it, even across function boundaries. With this any use of a global variable instantly leads to the inclusion of all defining-statements and their controlling structures. This may lead to drastically more dependencies but ensures correct control and data dependencies for global variables.

Another problem with generic C-programs is that there may be libraries present that have no source code to analyse except the function declarations of the library interface. Because it is in general impossible to obtain the information about SUMMARY-, PARAMETER_IN- and PARAMETER_OUT- edges from the function declarations one has to generate a safe standard representation for a library-call. In absence of further knowledge the most basic and safe assumption is that every argument passed to such library function is actually used and that the return value for a library depends on all arguments passed. From this follows that for a library-call the SUMMARY-edges fully connect every ACTUALIN with every ACTUALOUT. Even though this is usually not necessary it is the safest approach. With this extension it is therefore possible to process arbitrary C programs.

If the interface of the library is known, as it is the case of MPI, where there is a precise definition for every function parameter one can set the SUMMARY-edges accordingly. For MPI this has been demonstrated with MPI_Send and MPI_Recv. Unfortunately a SDG is to complex for even the simplest MPI-calls to visualize.

4.2.3 Slicing revisited

With a better understanding of these graphs it is now easily possible to create a slice of a program in regards to slice-target. For this one has to compute the set of nodes that are reachable starting from the source of the slice by exploring the system-dependence-graph in the appropriate direction, forward for a forward-slice and backward for a backward slice.

The computation of the backward-slice is based on the work of [7]. To obtain the backward slice of node x, which is the set of nodes that x depends on, the SDG is traversed in two passes with certain restriction for the edges being used in the traversal. In the first pass all edges except PARAMETER_IN edges may be used in the reverse traversal of the SDG. In this case the summary edges allow to pass a function-call without having to actually descend to the function definition. This corresponds to traversing upward in the call-graph toward the main function. For the second pass, one again starts to traverse the graph in the reverse direction, but this time CALL- and PARAMETER_OUT-edges may not be traversed. The final slice consists of all the nodes visited during both traversals.

With this slice-set of SDG-nodes the AST now can be pruned to remove any statements not included in the slice set, though one has to preserve the control-constructs that

the control-expressions are nested in. Additionally, one has to maintain syntactically correctness, such that a variable can only be used after it has been declared. Furthermore function-calls have to be specially processed to maintain the correct number of arguments passed. For this in a separate pass the parameters, that were not included in the slice, are added to the slice set and the dependencies for these nodes have to be satisfied again [7].

Back to the model-abstraction problem, slicing can now be used to remove any statements that have no influence on the MPI-calls found in the program. If one selects each MPI-call as the slice-target and merges the slice sets for each call, the remaining C-program consists of only the statements that support the MPI-calls.

4.2.4 Data abstraction

A further chance to reduce the complexity of a MPI-program is the fact, that most of the computation in such a program is usually focused to compute the data and not the envelope information. Since only the envelope and the number of elements send or received are of importance, it is possible to drastically reduce the complexity of the model by replacing the send or receive buffers of MPI-calls with MPI-SPINs abstract MPI_POINT type. Before such a replacement can be applied, it is necessary to check that no parameter used in the envelope section of any other MPI-call has a dependency-path to such a parameter. Since only receiving calls can possibly modify the content of the buffer only they require this analysis. To determined if the send-buffer is used in the envelope, one computes forward-slice of the receive-argument. If any node encountered during this traversal is a node marked as MODEL_TARGET, the abstraction can not be performed. A further observation, that scientific computations mostly focus on floating point calculations and most data send with MPI consists of such this type, supports this approach.

If the abstraction is possible, then one simply replaces the send- or receive-buffer with NULL and changes the type of the data transmitted to MPI_POINT without changing the number of elements send or received. The other parameters remain unchained, too. Because a forward-slice has been computed to determine the applicability, it is also possible to remove all non-control statements that data-depend on the specific ACTUALOUT-node. To maintain a valid abstract model, one has to replace the conditional-expression for control-statements with a random choice, because a valid evaluation is not possible anymore due to missing data-dependencies. The backward dependencies from the buffer of a send-call can easily be removed by the normal slicing-pass described earlier.

Even though automatic abstraction is technically possible and sometimes necessary, this has some noticeable impact on the precision of the model, which will possibly generate false-negatives due to the non-determinism of the random choices. Therefore this should only be used for variables that can not be modeled in PROMELA, like variables pointing to dynamic memory or floating point types.

4.3 C to PROMELA transformation

After the abstraction phase it is assumed that the program consists only of basic C-constructs with no calls to functions without a function body except MPI-calls. Such a program may consist of any control constructs of C, like *for*-loops, *if*-statements and any nesting scopes with curly brackets. Also a program at this stage may still contain data-types not directly supported in MPI-SPIN as long as the variables of these types are not used in the MPI-function-calls. Furthermore the program may only contain fixed size arrays with a limited number of elements, since this data is replicated in the state-vector causing a too large model. To generate more efficient models a threshold of 64 elements has been implemented to force data abstraction for such larger arrays. Even tough it is possible to construct multi-dimensional arrays in PROMELA by using typedefs, this was not implemented for this model extraction process and is therefore not allowed in the sliced program.

Finally, recursive function-calls may not be present in the abstracted program, because SPIN does not allow a stack and because some techniques used in the C to PROMELA transformation process can not resolve such recursions.

In this third phase of the model extraction process the program is transformed to the PROMELA-language.

4.3.1 Simplification of C

With these restrictions the remaining features not supported by SPIN are function-calls to known functions, the loops constructs, if and switch-expressions and floating-point types. All of these constructs can be transformed to a SPIN representation, either directly or encapsulated within C-blocks.

Furthermore global variables, though supported by SPIN, may not be present in the model. The reason for this is the fact that multiple instances of the main process will be instantiated during the verification. This causes multiple processes to access the very same variable, which usually would be in different process-spaces. This can be prevented by moving the variable into the main function. After switching such a variable to the main function, the scoping mechanism prevents further access from the other instantiations of the main-function in SPIN, but also for all functions originally using the global variable. Therefore one has to inline all functions that access this global variable into the main function to resolve this access problem. For this the existing inliner was used.

The remaining function-calls can be implemented in the model by calling the function in a C-code environment within PROMELA. One only has to correctly register all data-dependencies in the state-vector. Each function can be represented in the model by this method for as long as the called function does not contain any MPI-calls. If this is the case and if the containing function is called in a C-block the verification system would be completely unaware of these MPI-calls. Because no other mechanism can model this,

one also has to inline these functions to the main-function of the program to maintain a correct model, using again the available inlining-mechanism.

With the function-call problem solved, one has to transform all the control structures of C to either an *if*-selection structure or a *while*-structure in SPIN. As already stated, each selection construct in SPIN randomly decides which of all executable sequences it performs. This can be changed to a deterministic selection, if one simply provides only one such sequence together with the else part. In this case the selection has only a deterministic choice. With this it is now possible to describe the behavior of C's *while* and *if* constructs. To represent *for* and *do...while* loops, these loops have to be transformed to a while-loop first. Similarly one breaks up a switch-statement into multiple if-statements for a PROMELA representation.

Also the direct control transfer statements *break* and *continue* have to be adapted, because SPIN provides neither construct. This can be resolved by inserting an continue-label at the end of a loop and break-label after the loop. With these labels in place, one simply replaces the break with a goto to the break-label. The continue is handled in the in a similar fashion.

Finally it must be ensured, that all the control-expressions have no side effects, because they will be transformed later on to guards statements in PROMELA. If a control-expression has a side effect, like a function-call, a temporary variable is constructed to store the intermediate result of the expression. This boolean variable in then placed in the control-statement instead of the original expression. To maintain a correct value for that variable, the computation has to be inserted at the correct position in the program, to maintain correct behavior. For *if*-statements, this is directly before the *if* and for *while* such computations have to be inserted before the *while* and right before the end of the loop.

4.3.2 PROMELA transformation

So far the transformations have maintained an executable program, that is equivalent in its execution to the program after the slice.

With the following steps the source program is finally completely transformed to PROMELA. After these simplifications of the C-program the AST is traversed from the nodes upwards to the root, to ensure that all parents of marked children are also marked for the PROMELA transformation. To ensure that only once scope is used within a process-description, one has to collect all variables used in the main function and move them to the top most scope in the main function. If multiple variables are found, that have the same name, each one is assigned a unique name and all instances of that variable are replaced with that name. At the same time one can test, if the type of each variable is unsupported. If the type is not supported, the variable must be added to the state-vector using the c_decl-statement.

Because SPIN has no notion of return values there is also no return statement available. Since a SPIN process terminates when it reaches the end of the process-definition, it

suffices to jump to that end to indicate a normal termination. This is implemented by inserting an additional end-label at the end of the main function and replacing every return-statement in the main function with a *goto* to that end label. In addition the expression that is possibly nested in the return statement has to be inserted before that goto which replaced the return.

After these transformations the AST-nodes that are the source of control-dependencies for the MODEL_TARGET marked nodes, are marked for the direct transformation to PROMELA. All other nodes have only data dependencies to the MODEL_TARGETs and can be safely encapsulated in C-blocks.

Before the AST can be unparsed to PROMELA, the references of variables used in C-blocks that access the state-vector have to be adapted transformed to correctly access this data. The state-vector is accessed in C-code by deferring the predefined variable, with the name P and the name of the process. For the main process this is Pmain. Therefore the variables-access to the state-vector variable A of process main would look like Pmain−>A in a C-block.

After all these transformations and modifications the AST consists of nodes that either have been marked for a direct transformation to PROMELA and those who have not been marked.

The code generation of this transformed AST could be achieved with a few modifications to the standard C-unparser of ROSE, due to the similarities of C and PROMELA. For this every statement, that has not been marked for the PROMELA-transformation is surrounded by an .c_code-statement during the unparsing phase. The only statements that had to be treated differently where the if and the *while*-statement, which unparsed either to the PROMELA version if the node was marked for the PROMELA-transformation or else unparsed normally to their C-representation.

4.4 Summary

This chapter introduced and described the steps and algorithms necessary to extract the MPI-communication model from a C-program. Even though a fair amount of restrictions for the original C-code exists, this process can be applied without any further user interaction. A fair amount of work was invested to implement the necessary analysis to enable this process. The system dependence graph was shown to be a very versatile and useful representation of the control and data structure of a program. Further implementation of different analyses in ROSE probably will enable a more precise construction of the structures that the model extraction process is based on.

5 Evaluation

This chapter will describe the steps and methods that were applied to test the model-extraction process for its ability to generate models for the communication structure. For this it should be clear that it is impossible to fully test whether the extracted model can satisfy any imaginable application. On the other hand it should be possible to select a few core features which the abstraction process should be able to satisfy. For automatic extraction of the communication model the first step is to test, if the generated model-description matches the input requirements of the chosen model checker. In this case, the generated model should at least satisfy the constraints of SPIN, which means that it must be a part of the extended PROMELA-language. This is automatically tested by using MPI-SPIN to verify any extracted model. If the input is rejected, the model is not a part of PROMELA.

One would also expect, that such a model represents the communication of the original program. This is difficult to test, because no analysis of the model can answer, if the model is precise enough, without knowing the properties. This can only be said about the original program. Therefore the five basic security properties for MPI-programs which are defined in[20] were chosen as a guideline for the quality of the model. Fortunately MPI-SPIN automatically verifies those properties. These fundamental properties of MPI, freedom of deadlocks, freedom of intersecting send and receive buffers, limited number of MPI-requests, freedom of outstanding requests after finalize and buffer-overflows for out-parameters, can not be tested exhaustively because only a limited number of MPI programs exists. Furthermore, the more complex the program, the more obscure are model-representations and the more difficult becomes an evaluation, which limits the size for manual checks.

For the reasons described above, one would test the extraction process and the validity of the extracted models for small, well defined programs that cover the important aspects of C and MPI. If all these tests show no errors, this is an indication that the composition of the tested features will also be valid. These tests should contain function-calls, either with MPI or without, different loop and control constructs, different data types and arrays smaller and larger than the predefined boundary. For the MPI there are too many different functions to be tested. For this work at least one representative MPI-function of the blocking, non-blocking and group-communication methods has been tested. These tests consisted of creating a valid MPI program, for which the extraction process was performed. After the confirmation of correctness of the model by MPI-SPIN, the source program was modified to contain errors that the model checking system had to discover.

5.1 Initial tests

The first tests are based on a typical Hello-World MPI-program (Lst. 5.1)[1]. The tests performed on this program were designed to check the basic transformation capability of the developed tool and the interaction with MPI-SPIN. The program was successfully transformed into the model (Lst. 5.2)[1] and was verified to satisfy the basic security properties for 2 and more processes.

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int         numprocs, myid, buf, i;
  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  printf("I'm number %d of %d.\n", myid, numprocs);
  if (myid==0) {
    buf=1;
    for (i=1; i<numprocs; i++) {
      MPI_Send(&buf,1,MPI_INT,i,0,MPI_COMM_WORLD);
    }
    printf("Proc %d: Message sent to %d processes.\n", myid,i-1);
  }
  else {
    MPI_Recv(&buf,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
  MPI_COMM_WORLD, &stat);
    printf("Proc %d: Received data %d from %d with tag %d.\n",
        myid, buf, stat.MPI_SOURCE, stat.MPI_TAG);
  }
  MPI_Finalize();
}
```

Listing 5.1: First MPI test program

After this first run the tag-information for the receiving process was modified such that it would not receive any message from the sending process, therefore forcing it to deadlock. This was automatically discovered by SPIN. The next test performed with this code involved the alteration of the send variable to an array of 50 elements

[1]Further test programs will be given in a simplified and reduced form, with only the necessary MPI functions and the surrounding control constructs stated

[1]This model was manually indented for a better presentation

```
#include "mpi-spin.prom"
#include "mpi-collective.prom"
c_decl{typedef enum{false=0,true} bool;}

active [NPROCS] proctype main()
{
MPI_Status stat;
int i;
int buf;
int myid;
int numprocs;
MPI_Init(Pmain,Pmain -> _pid);
c_code
{
  Pmain -> numprocs = NPROCS;
  Pmain -> myid = RANK(Pmain);
};
if
::(c_expr{Pmain -> myid == 0})->
{
  c_code{Pmain -> buf = 1;};
  c_code{Pmain -> i = 1;};
  do
  ::(c_expr{Pmain -> i < Pmain -> numprocs})->
  {
    MPI_Send(Pmain,((&Pmain -> buf)),1,(MPI_INT),Pmain -> i,0);
    c_code{Pmain -> i++;}
  }
  ::else break;
  od;
}
:: else ->MPI_Recv(Pmain,((&Pmain -> buf)),1,(MPI_INT),
    MPI_ANY_SOURCE,MPI_ANY_TAG,&Pmain -> stat)
fi

MPI_Finalize(Pmain);
}

active MPI_PROC
```

Listing 5.2: The automatic model for (Lst.5.1)

with the respective adaption of the count parameter in the source file. Then again the verification process was started for the model, which yielded the expected confirmation of correctness. When the number of elements that the receive should retrieve was increased beyond the size of the array, the verification system discovered truthfully an error. Further modifications to the basic program, like omitting the MPI_Init, were also always discovered. These tests were performed also with arrays beyond the limit of 64 elements, with floating points types and arrays of floating point types, all with the same successful results. Therefore for this small sample, one can assume, that the extraction process is correct.

5.2 Further basic tests

The same program was rewritten using the non-blocking MPI_Isend and MPI_Irecv together with MPI_Wait (Lst. 5.3), using a barrier (Lst. 5.4) and all-reduce (Lst. 5.5) collective operations with no changes in the results. For each of these programs the same alterations used during the initial test were performed and detected during verifications. With these encouraging results the tests could be extended to more difficult programs.

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int         numprocs, myid, buf, i;
  MPI_Status stat;
  MPI_Request    request;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);;
  if (myid==0) {
    buf= ... some computiations ...;
    for (i=1; i<numprocs; i++) {
      MPI_Isend(&buf,1,MPI_INT,i,0,MPI_COMM_WORLD,&request);
      ... some computations ...
      MPI_Wait(&request,&stat);
    }
  }
  else {
    ... some computations ...
    MPI_Irecv(&buf,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPI_COMM_WORLD,&request);
    ... some computations ...
    MPI_Wait(&request,&stat);
```

```
  }
  MPI_Finalize();
}
```

Listing 5.3: Pseudo-code for the program used to test non-blocking MPI communication

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int        numprocs, myid, buf, i;
  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (numprocs<2)
  {
  MPI_Finalize();
  return 0;
  }
  buf= ... some computations ...
  if (myid==0)
  MPI_Send(&buf,1,MPI_INT,i,0,MPI_COMM_WORLD);
  ... some computations ...
  MPI_Barrier(MPI_COMM_WORLD);
  ... some computations ...
  if (myid==1)
    MPI_Recv(&buf,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPI_COMM_WORLD,&stat);
  ... some computations ...
  MPI_Finalize();
}
```

Listing 5.4: Pseudo-code for the program used to test barrier operations

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int        numprocs, myid, buf, i;
  MPI_Status stat;
  MPI_Init(&argc,&argv);
```

```
  MPI_Comm_size(MPLCOMM_WORLD, &numprocs);
  MPI_Comm_rank(MPLCOMM_WORLD, &myid);
  int out,in=1;
  MPI_Allreduce(&out,&in,1,MPI_INT,MPI_SUM,MPLCOMM_WORLD);
  MPI_Finalize();
}
```

Listing 5.5: Pseudo-code for the program used to test collective MPI communication

5.3 More complicated programs

After these preliminary tests further sample programs were used to examine the extraction mechanism for more complicated programs. The abstraction of these programs was usually quite successful. However, the data abstraction of large arrays and unsupported types cause some problems during verification due to false-negatives.

The test-program (Lst. 5.6) is a good example for this.

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int        numprocs, myid, i;
  float buf;
  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPLCOMM_WORLD, &numprocs);
  MPI_Comm_rank(MPLCOMM_WORLD, &myid);
  printf("I'm number %d of %d.\n", myid, numprocs);
  int nextRank=(myid+1)%numprocs;
  if (myid==0) {
    buf=500;
    MPI_Send(&buf,1,MPI_FLOAT,nextRank,0,MPLCOMM_WORLD);
    MPI_Recv(&buf,1,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPLCOMM_WORLD,&stat);
  }
  while(buf>0)
  {
    buf-=5.0;
    MPI_Send(&buf,1,MPI_FLOAT,nextRank,0,MPLCOMM_WORLD);
    MPI_Recv(&buf,1,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPLCOMM_WORLD,&stat);
  }
  for (i=0;i<numprocs;i++)
  {
```

```
    if (i!=myid)
    MPI_Send(&buf,1,MPI_FLOAT,i,0,MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

Listing 5.6: Test program for data abstraction

Because SPIN does not support floating point data types as argument in MPI-calls, all floating point variables had to be removed my means of abstraction. To maintain at least a coarse representation of the program the control-expression of the while loop and the second if-statement had to be replaced by an random decision. Because of this non-deterministic behavior the verifier could terminate the loop at any time and cause the program to deadlock. Unfortunately a complete model could not be extracted for this problem, though some true-errors could be found with this model without encountering a false-negative first.

5.4 Problem with function calls

Further tests with abstracting function calls (Lst. 5.7) showed, that the exiting inlining-transformation generated not a completely correct AST for the given functions. Detailed testing showed that the transformed AST correctly represented the inlined program, though some technical requirements of the AST were faulty. To continue testing the transformation of this program, the AST was temporarily unparsed to source code and re-read to fix the errors in its representation. After this the program could by completely transformed and successfully verified. This problem is not based in the presented approach but in the inliner and can be explained by the fact that ROSE is still under development.

```
#include "mpi.h"
#include "stdio.h"
int master_dist()
{
  int nprocs;
  int sendData;
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  for (int i=0;i<nprocs;i++)
  {
    sendData=i;
    MPI_Send(&sendData, 1, MPI_INT, i,1, MPI_COMM_WORLD);
  }
  return 0;
}
int compute()
```

```
{
  int limit;
  MPI_Status computeStatus;
  MPI_Recv(&limit, 1, MPI_INT, 0,1, MPI_COMM_WORLD, &computeStatus);
  int sum=0;
  for (int i=0;i<limit;i++)
  {
    sum+=i;
  }
  MPI_Send(&sum,1,MPI_INT,0,2,MPI_COMM_WORLD);
  return 0;
}
int master_collect()
{
  MPI_Status status;
  int nprocs1;
  int collect;
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs1);
  int sums;
  for (int i=0;i<nprocs1;i++)
  {
    MPI_Recv(&collect, 1, MPI_INT,MPI_ANY_SOURCE,2,
        MPI_COMM_WORLD, &status);
    sums+=collect;
  }
  return sums;
}
int main(int argc, char ** argv)
{
  MPI_Init(&argc,&argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank==0)
  {
    master_dist();
  }
  compute();
  if (rank==0)
  {
    master_collect();
  }
  MPI_Finalize();
  return 0;
```

```
}
```

Listing 5.7: Program for tests of function calls

5.5 A complex example

The evaluation was concluded by a final test. For this test, the developed extraction process was applied to a parallel matrix-matrix multiplication from [21]. For this MPI-program (Lst. 5.8) the communication-model was also successfully abstracted and the model verification system could again be applied to check for the basic security properties with MPI-SPIN. This program was tested with different sizes for the matrices and different sizes for the grid. Again different modifications which would have caused invalid communication behavior were applied and successfully caught by the verification system.

```
#include "mpi.h"
#define N 4
#define M 4
#define L 4
int main(int argc, char ** argv)
{
  double A[N][L],B[L][M],C[N][M];

  int rank,nprocs,i,j,numsent,sender,row,anstype;
  double buffer[L], ans[M];
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank==0) { /* I am the master */
    numsent=0;
    for (i=0; i<nprocs-1; i++) {
      for (j=0; j<L; j++)
        buffer[j] = A[i][j];
      MPI_Send(buffer, L, MPI_DOUBLE, i+1,
          i+1, MPI_COMM_WORLD);
      numsent++;
    }
    for (i=0; i<N; i++) {
      MPI_Recv(ans, M, MPI_DOUBLE, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      sender = status.MPI_SOURCE;
      anstype = status.MPI_TAG-1;
      for (j=0; j<M; j++)
        C[anstype][j] = ans[j];
```

```
      if (numsent<N) {
        for (j=0; j<L; j++)
          buffer[j] = A[numsent][j];
        MPI_Send(buffer, L, MPI_DOUBLE, sender,
            numsent+1, MPI_COMM_WORLD);
        numsent++;
      }
      else MPI_Send(buffer, 1, MPI_DOUBLE, sender,
          0, MPI_COMM_WORLD);
    }
  } else { /* I am a slave */
    while (1) {
      MPI_Recv(buffer, L, MPI_DOUBLE, 0,
          MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      if (status.MPI_TAG==0) break;
      row = status.MPI_TAG-1;
      for (i=0; i<M; i++) {
        ans[i] = 0.0;
        for (j=0; j<L; j++)
          ans[i] += buffer[j]*B[j][i];
      }
      MPI_Send(ans, M, MPI_DOUBLE, 0,
          row+1, MPI_COMM_WORLD);
    }
  }
  return 0;
}
```

Listing 5.8: Matrix multiplication program

6 Summary and future prospects

6.1 Related work

A comparison with related work in the model-checking community showed that this work provides one of the first approaches to fully automatically extract the MPI model from a C-program for MPI-SPIN, even though in a limited form. Other software verification projects and tools, like the ModEx-tool [11], Java Path Finder [26] and the SLAM [4] can not perform the extraction completely automatically or do not provide the capabilities to model the MPI communication. All these tools provide some form of partial automated model extraction and verification but always require some user input specifying what the goal of the verification is.

The ModEx-tool, which was developed in conjunction with SPIN [12] employs a simple control flow analysis to compute the control statements in SPIN. The remainder of the program is encapsulated complete in c-blocks without any further automatic analysis. The user can specify abstractions by populating a lookup-table which the ModEx-tool uses to directly textually replace some parts of the program. Even though this is a very simple approach it has been used successfully to verify a commercial telecommunication system [12]. This approach can not be used to completely automatically generate a model of MPI communication due to its lack of data-flow analysis and automatic abstraction capability.

The Java Path Finder [26], called JPF, in contrast is a well designed verification system for Java programs. It uses a special developed Java-virtual machine to perform an explicit state verification, similar to SPIN. Therefore, it is bound to the same limitations and can only verify pure Java code, without any system-interaction. Besides slicing to reduce the programs complexity JPF also provides the user with an interface to specify predicates. These predicates are used to generate a new Java program that operates on the more abstract predicates, which is then verified. This system can only be applied to languages that can be transformed to Java byte-code. Furthermore there is no MPI-verification available at this time.

The last mentioned system, the SLAM-project [4], is a verification system used to verify the usage of APIs. This specific verification system generates an abstract boolean program from the input and uses the BeBop [3] model checker as a back end for verification. The abstract boolean program is constructed from the control-flow of the source and from predicates specified by the user. These predicates, in combination with the predicates from the source code, allow a transformation for variable-uses to a boolean

value, depending on the evaluation of the predicate. Because of the boolean nature of the model, this approach does not support the representation of MPI integer based envelopes and can not be used for model-checking of MPI programs.

6.2 Summary

The goal of this work was to develop a process to automatically verify the use of MPI in an arbitrary C-program. Before any such process can be addressed one has to define the bounds in which such a system may operate, because a general approach is impossible, as with the Turing-problem. Therefore, this work restricts itself to the extraction of the MPI communication. For this the MPI-SPIN verification system was selected as the target platform. Besides providing the necessary MPI verification capability this verification system has some rather unique support for automatic model extraction that was used in this work.

The core problem of this work was to acquire and implement the tools for constructing an as small as possible PROMELA representation of the MPI communication for a C-program. For this a system dependence analysis [7, 18, 9, 16] was implemented for the ROSE-framework. To be able to process MPI programs, this analysis had to be extended to handle library calls. With this graph an abstraction technique, called slicing, was implemented to reduce the size of the program. Furthermore a data abstraction transformation, based on slicing, was developed to handle data types not supported by MPI-SPIN. The final transformation of the source program to PROMELA was handled using different existing transformations already available in ROSE. For the final output of the PROMELA code the existing unparser was modified to handle the slight differences between C and PROMELA.

To enable this completely automatic model extraction several restrictions to the original program had to be made. Since many data analyses have problems to handle aliases of variables correctly, it was required that the initial program must not contain aliasing. Also, due to the restrictions enforced by the verification system, the MPI calls in the program can not depend on any non-MPI library calls after the slicing step. This is especially true for systems calls, like IO operations. The restriction to global communication derives form the missing implementation of communication groups for MPI, so that programs using group communication can not be modeled.

Tests of the tool showed that for programs where the MPI calls solely depend on control and data types supported by SPIN a very usable model can be extracted. If the model extraction process has to abstract data types and is forced to replace previous deterministic control statements with non-deterministic ones, the model becomes unstable. Such a model may possibly generat false-negatives during verification.

6.3 Outlook

Future work on this model extraction process may utilize more compiler analyses to improve precision for the system dependency graph. For example one could use an improved pointer analysis to track memory locations. A further step could then transform these memory locations to normal variables. Once transformed, it would be possible to even extract the control information stores in larger arrays and to prevent over-abstraction of arrays.

An other possible approach for improving the automatic model extraction would be, to create annotations for often used library functions, like the mathematical functions min, max and square root. With this information one could allow such calls and encapsulate them in C-code statements for even greater flexibility of this automatic extraction approach.

Further improvements to the given approach might even provide the user with annotation mechanisms similar to SPF [26] to guide the automatic abstraction of a program. With these annotations a user could provide hints about which data representation should be used for variables. For example the user could indicate for a floating point variable a complete abstraction, as it is done with the current approach, or select a symbolic approximation or even a mapping to integers. With such annotations the user also could provide limits for recursions depths enabling the inlining of such constructs. Many more possible techniques might be possible, which remain to be discovered by future works.

At this point also a few suggestions for some modification of the SPIN verification system are proposed. Even tough the current version of SPIN provides mechanisms to support model extraction, this process requires further improvement to support the abstraction of higher languages like C. For example, it is not possible to annotate the code with information where a statement was extracted from and what its source-form looked like.

This is especially difficult for the evaluation for error trails, because such a trails describe the SPIN statements and not the original C-code. A user therefore has to manually compare the abstract model with the original program to be able to understand the error. This could be supported to a great extend, if PROMELA had a support mechanism to indicate source positions similar to the debug-symbols in binaries.

A problem with the current version of SPIN is, that if one discovers false-negative one has no direct means to indicate to the verification system to ignore the same error trail for a consecutive evaluation. It would greatly assist the user if one could do so, because this would enable a more comfortable application of model checking.

Acknowledgments

I like to thank Dan Quinlan and his team from the ROSE-Project for the opportunity to work with them. This was very pleasing and challenging experience.

I like to thank Professor Pflaum and Jochen Hrdtlein for making this work possible.

For all the discussions and advices I would like to thank Silke Bergler, Andreas Saebjornsen, Ramakrishna Upadrasta, Jeremiah Wilcock, Thomas Panas, Stefan Iwainsky and Brian Taylor, you were a lot of support.

Furthermore I would like to thank the Lawrence-Livermore-National-Laboratory and the staff of ISCR for their support. Without them, I would not have been able to accomplish this work.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[3] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

[4] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[5] Robert A. Ballance and Arthur B. Maccabe. Program dependence graphs for the rest of us. Technical report, The University of New Mexico, Department of Computer Science, Jan 16 1992.

[6] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, Berlin, Germany, 2001.

[7] David Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2(1–4):31–45, March–December 1993.

[8] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[10] Peter R. Glück and Gerard J. Holzmann. Using spin model checking for flight software verification. In *Aerospace Conference Proceedings, 2002. IEEE.*

[11] G. J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, 28(4):364–377, April 2002.

[12] Gerard J. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, MA, USA, 2003.

[13] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tennessee, Knoxville, TN, June 1995.

[15] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Fransisco, California, USA, 1997.

[16] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.

[17] Daniel J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[18] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, New York, NY, USA, December 1994. ACM Press.

[19] Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.

[20] Stephen F. Siegel. Model checking nonblocking MPI programs. In *Lecture Notes in Computer Science: Verification, Model Checking, and Abstract Interpretation.*

[21] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In Lori L. Pollock and Mauro Pezzè, editors, *ISSTA*, pages 157–168. ACM Press, 2006.

[22] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference. Volume 1, The MPI-1 Core*. MIT Press, Cambridge, MA, USA, second edition, September 1998.

[23] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*.

[24] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (German version)*. Pearson Studies, Munich, Germany, 2003.

[25] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Procedings of the London Mathematical Society*, 42(2):230–265, 1936.

[26] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.