Comparing the LTSmin and NuSMV reachability tools via automatic translation of their respective input languages

MSc Thesis

Daan van Beek
University of Twente,
The Netherlands,
d.i.vanbeek@student.utwente.nl,

Supervised By:
Prof. Dr. J.C. van de Pol,
G. Kant MSc,
Prof. Dr. Ir. A. Rensink

August 6, 2013

Preface

The Thesis that lies before you is a Final Project report. The Final Project is a project (30EC) which is completed as the last step to receiving a master's degree. In this case, the sought after master degree is the Computer Science - Formal Methods and Tools degree as taught at the University of Twente in the Netherlands. Prior to this Final Project the Research Topics project (10EC) has been completed which aims to identify the contents of the Final Project. In the Research Topics project the goals of the Final Project where determined. Research was done under the broader title of "Comparing LTSmin and NuSMV". Soon it was determined that the comparison would be between their respective symbolic reachability tools. The main focus of the Research Topics project was to figure out how to compare the LTSmin and NuSMV tools as they do not support the same input languages and thus cannot be run on the same input models in order to compare their execution times. As a reaction to this question it was determined that the chosen method of comparison would be the creation of automatic translations between their respective input languages.

Acknowledgements

I would like to thank Jaco van de Pol, Gijs Kant and Arend Rensink for their invested efforts and guidance during my Final Project.

Contents

Chapter 0

Introduction

Today's life standards entail that the average human depends more and more on safety-critical systems. Verification of such safety-critical systems keeps the quality of those systems high as to prevent major disasters or other disruptions of daily life. One such verification technique is called model checking which attempts to create a model of the safety-critical system such that the model can be checked for its correctness. Any faults found in the model are investigated to see if the fault also exists in the modelled system. Model checking tools are able to check if certain properties hold on the created model or not. The analysis of models is often costly in time. Comparing model checking tools and explaining the obtained results might provide insight on how to increase the performance of the model checking tools.

NuSMV, a symbolic model checker, and LTSmin, a symbolic model checker toolset both have tools that can compute the reachable state space of a model. We have chosen to compare NuSMV and LTSmin's reachability analysis tools as reachability analysis is an important part of any model checking activity. The NuSMV and LTSmin reachability tools are currently incomparable as they do not support the same input languages. mCRL2 is one of the modelling languages supported by LTSmin and NuSMV supports the SMV language. Comparing the performance of the NuSMV and LTSmin reachability tools is interesting because they both use different techniques: NuSMV uses "conjunctive paritioning" and LTSmin uses "disjunctive paritioning" (both techniques are explained in the preliminaries chapter). Another important difference is that NuSMV uses an input language that directly represents the needed transition relation whereas LTSmin has to learn the transition relation from a description that is not a direct representation. The research described in this Thesis attempts to compare the NuSMV with the LTSmin reachability tool by providing automatic translations from the SMV language to the mCRL2 language and experimentally executing the original and translated models on their respective platform.

A multitude of possible automatic translations were identified that translated both from languages that are supported by LTSmin to the SMV language (the input language of NuSMV) and the other way around. An existing translation from PROMELA (an LTSmin input language) to SMV was identified to be available in the P2B and S2N tools [2, 18]. Later, in the continuation of the project it was determined to focus on translations from SMV to LTSmin languages as no research was found on such translations. mCRL2 was chosen as an appropriate LTSmin input language as it supports language constructs that enable convenient translations from SMV models to mCRL2 models.

In order to translate from SMV to mCRL2 a two phase approach was adopted: SMV is first translated into SMV-flat, which is a flattening translation as is also used in the SMV toolset. The SMV to SMV-flat translation translates any SMV model with an arbitrary number of modules (synchronous modules, asynchronous modules) into one big synchronous SMV model. This makes translation to mCRL2 easier as all SMV models are first translated to this flattened format. Having an SMV-flat model, we translate to mCRL2 in one of two ways, the first being the All-in-1 translation which translates each state in an SMV model to a single state in the translated mCRL2 model. The second translation is called 1-by-1 which has multiple mCRL2 states for each single SMV state.

Going back to the bigger picture, in this thesis we:

- Provide automatic translations from SMV to an LTSmin input language.
- Formalize and prove these translations to be correct.
- Use the translations to translate several SMV models to mCRL2 models and execute them on SMV and LTSmin in order to obtain experimental results.
- Explain the results by using the theory behind the NuSMV and LTSmin reachability tools.

The rest of this document is divided into several chapters: Chapter 1 explains the theory behind the SMV and LTSmin reachability tools. Chapter 2 focusses on the created and implemented translations, the formalization of the input languages and the formalization and correctness proofs of the translations. Chapter 3 analyses possible SMV models to use, shows the results of their execution on the reachability tools and explains those results using the theory described in Chapter 1. Chapter 4 reports on the main conclusions and advising on future work. Chapter 5 concludes the report with a short description of the developed translation tool.

Chapter 1

Preliminaries

This chapter discusses the theoretical background of this project. It discusses what model checking is, how it is performed and what model checking algorithms are used by LTSmin and NuSMV. It discusses both sequential as symbolic model checking and possible techniques to partition the transition relation. This chapter also explains about LTSmin's special features and which features are used by the NuSMV toolset. The difference between synchronous and asynchronous models and the way they handle parallel composition is also described.

1.1 Model checking

The problem solved by model checking has a model and a property as its input. The output of the problem is a yes or no answer with possibly a counterexample to the question: "Does the property hold is this model?". A model defines a number of data variables together with a descriptions of how those data variables may change called next state valuations. The data variables are denoted $x_1, ..., x_N$, with N the number of variables defined in the model. Each possible assignment of values to all data variables amounts to a single valuation of those data variables called a "state". The individual values of the data variables $x_1, ..., x_n$ for a certain state are denoted $v_1, ..., v_n$. Models are described by transition systems:

Definition 1 (Transition System)
A transition system (TS) is a structure $\langle S, R, s^0 \rangle$, where S is a set of states, $R \subseteq S \times S$ is a transition relation and $s^0 \subseteq S$ is the initial state [5]. The set of next states of a state s, is defined to be $\{s' \in S \mid R(s, s')\}$.

Model checking for invariants, such as "Property x may never hold", boils down to a reachability analysis of the state space of the transition system. This reachability analysis will either find a state in which property x holds, or conclude that such a state does not exist. The reachability problem has as input a set of initial states and a transition relation. The solution of the problem is a description of all reachable states, that is: states that can be reached by applying the transition relation zero or more times to one of the initial states as defined in definition 2.

Definition 2 (Reachable States)
Given a transition system TS = $\langle S, R, s^0 \rangle$, The set of reachable states is $V = \{s \in S \mid s^0 R^* s\}$. A state $s \in S$ is reachable if $s \in V$ [5]. R^* stands for the application of R zero or more times.

1.1.1 Enumerative model checking

When using enumerative model checking the state space is searched one state at a time. If a state is found then all its successors are computed and put in a list that signifies that they are new and should be investigated later. There is also a list of visited states which is used to find out if the new states are not investigated already in order to detect loops and prevent unnecessary work and a never ending search. The paper by Blom and van de Pol (2008) provides a simple algorithm in Table 1 [5], reproduced here:

```
1  proc reach()
2    V := {s^0}
3    L := V
4    while L ≠ ∅ do
5      L := {y | ∃x ∈ L : x R y}
6      L := L \ V
7      V := V ∪ L
8    end
9    return V
10 end
```

Figure 1.1: A pseudo-code algorithm for a breadth-first search for all reachable states.

This algorithm describes a breadth-first search for all reachable states starting from state s^0. The algorithm starts by adding the initial state to the set of visited states V and by also adding the initial state to the set L of states that have been discovered but not yet expanded (their successors have not yet been investigated). Then we enter a while loop that first expands all states in L by sequentially expanding each state in L. Expanding a state means that all successor states of that state are discovered by applying the transition relation R. After all states in L have been expanded, all states in V are subtracted from the set L. This is done in order to detect loops in the transition system. If a loop is present, then we do not want to expand states that already have been expanded as we would then enter an infinite loop. The last statement in the while loop makes sure that the visited set is updated for this round. The while loop stops when there are no more new states to be expanded, in other words, no new states where found in the last round and we have thus explored all reachable states which are now recorded in set V.

The storage requirements for the list of visited states is a major drawback of enumerative model checking. Often advanced hash tables, such as Bloom filters [12], and Cleary tables [11], are used to keep the storage requirements of the list of visited states to a minimum. A problem with these advanced hash tables is that these methods often sacrifice completeness. Other solutions are available that do not have such a drawback [14]. Enumerative model checking also has the drawback that it has to look at each state separately when expanding a group of states, which is costly in time.

1.1.2 Symbolic model checking

Symbolic model checking is a solution to these storage requirements and time problems. The core of symbolic model checking is a data structure called a Binary Decision Diagram (BDD). The advantage of

BDDs is that a set of states (or a transition relation) can be represented in a compressed form. An even bigger advantage is that a BDD representing a set of states or a transition relation can be used in set operations without decompressing the information first. Information compression by BDDs does not incur any kind of information loss. [1, 7] give descriptions on BDDs. [17] gives a mathematical description of the symbolic model checking process using BDDs, [5, 8] give a more practical explanation.

Binary Decision Diagram

A BDD [1] is a directed acyclic graph (DAG) with zero or more internal nodes, and special leaf nodes. The special leaf nodes in the diagram and are depicted by squares with the labels "1" and "0", representing true and false respectively. The internal nodes represent boolean variables. Each internal node has two outgoing edges, one normal and one dashed edge respectively called the true, and the false edge. One node of the BDD is seen as the root of the BDD. This root node has no incoming edges. A path in the BDD staring at the root node and ending at one of the special leaf nodes defines a valuation over all variables in the BDD. The path also tells us whether the valuation represented by this path is in the represented set or not. If the path ended in a leaf labelled "0", then this valuation is not in the set represented by the BDD. If the path ended in a leaf labelled "1", then this valuation is in the set represented by the BDD. Let us clarify the idea of a valuation by giving a definition:

Definition 3 (Valuation)
A valuation over a set of boolean variables, denoted $v_1, ..., v_n$, is an individual assignment of one of the values {0,1} to each of the boolean variables. A set of x boolean variables has 2^x different valuations.

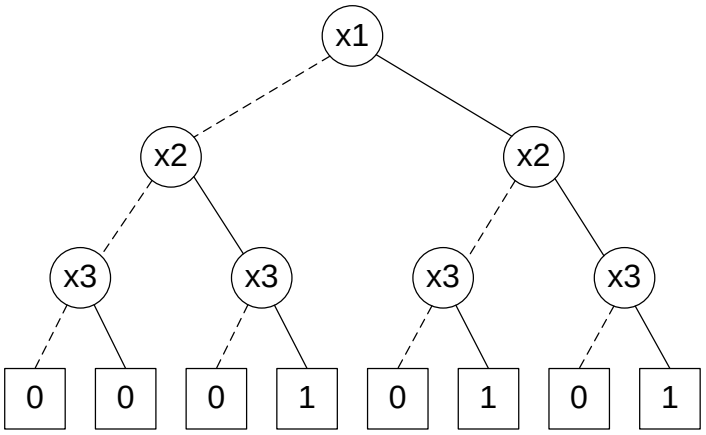


Figure 1.2: An example of a BDD [7].

Let us look at an example. Figure 1.2 shows a BDD that has three internal nodes labelled x_1, x_2 and x_3, and multiple leaf nodes labelled 0 and 1. The nodes x_1, x_2 and x_3 represent three boolean variables. The paths through this BDD from the root node x_1 to the leaf nodes $\{0, 1\}$ represent whether certain valuations over the variables x_1, x_2 and x_3 are in the represented set or not. For example, the valuation $x_1 = false, x_2 = false$ and $x_3 = false$ is not in the set. We can see this by traversing the corresponding

path in the BDD: Staring at the root node x_1 we see that the choice of $x_1 = false$ leads to the left x_2 node (by following the dashed edge). The choice that variable x_2 is also false leads to the leftmost x_3 node and the choice that x_3 is false leads to a special leaf node 0. As the path belonging to valuation $x_1, x_2, x_3 = false$ leads to a special leaf node labelled 0, this valuation is not in the set represented by the BDD. If we however change the valuation to assign true to variables x_2 and x_3, then the traversal of the BDD changes as we now choose the normal (non-dashed) edge in node x_2 and afterwards do the same in node x_3 with the effect that we end up in a special leaf 1. This indicates that the valuation $x_1 = false, x_2, x_3 = true$ is in the set represented by this BDD.

BDDs are used in symbolic model checking to represent sets of states and or the transition relation. Normally a state consists of a number of data variables, with types such as boolean, integer, string etc. BDDs only work with boolean variables and therefore variables with another type must be represented by multiple boolean variables. In general this is always possible as computers themselves also represent all data with "boolean" variables: the "bit". Recursive data types however present problems as they can become arbitrary large. Regardless researchers have been able to find ways to symbolically represent such recursive data types [5].

Once all variables have been transformed to boolean variables it becomes clear how BDDs can be used to represent for example the set of visited states: Say that a models state can be described by N boolean variables. Then each possible state of the model (reachable or not) is represented by a single valuation over this set of boolean variables. If we create a BDD that has a path for each possible valuation, we can encode the set of visited states into this BDD by lettings paths representing visited states reach the special "1" leaf node. Paths that represent states that have not been visited are going to the 0 leaf node indicating that they are not in the set of visited states. Formally we define the relation between the BDD representation and the set of visited states as follows.

Relation 4 (Relation Between a BDD and its Represented Set)
Let D denote a BDD. Let M be a model who's states are described by n boolean variables denoted x_1,...,x_n. Let state s have a valuation $v_1, ..., v_n \in \{0, 1\}$. Then s is in the set of visited states represented by D iff D has a path constructed by following the corresponding true or false edge to valuation v_1 for node x_1,..., v_n for node x_n which leads to a special leaf node 1.

The reason why BDDs and symbolic model checking is used in practice is three fold. Firstly BDD representations of a set of states generally use much less memory then an explicit list of the states. The reason for this is that in a BDD lots of valuations can share data, whereas an explicit list would record the same data multiple times. Data sharing in BDDs is enforced using multiple BDD simplification methods which transform an ordinary BDD into a Ordered Binary Decision Diagram (OBDD) [7] which effectively removes redundant data. Figure 1.2 can also be simplified. The simplified version is shown in Figure 1.3 which essentially represents the same information but uses much less nodes (check the paths in the BDDs to see the equivalence between the two BDDs). The absence of a node means that the valuation of that variable along the current path is irrelevant. Both a true and false evaluation have the same effect on the continuation of the path. For example, if variable $x1$ is evaluated to true, then the valuation of variable $x2$ does not matter. For both possible valuations of variable $x2$ we need to look at variable $x3$ to see if this valuation is in the represented set or not.

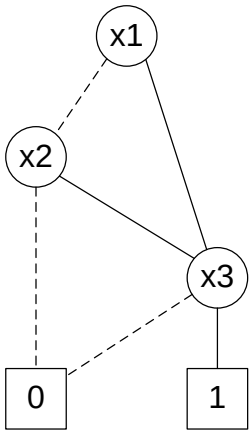


Figure 1.3: An OBDD representation of Figure 1.2.

Secondly, BDDs can be used with boolean set operations without decompressing the information first. It is possible to compute the boolean AND of a BDD representation of a set of visited states with a BDD representation of a transition relation without decompressing either BDD data structures, and thus effectively computing the next state function (explained in the next section). Thirdly, BDD operations are set operations, meaning that all operations compute their operation over the complete set of valuations represented by the BDDs at once. The next section describes how the reachability analysis works using BDDs.

Reachability analysis with BDDs

BDDs can be used in model checking by having one BDD that represents the visited set, and one BDD that represents the transition relation. The variables used in the visited set are boolean representations of all variables that describe a single state. The vector of variables used to describe a single state is denoted x, with $x = x_1, ...x_n$ and is called the "state vector" The visited set BDD is denoted $V(x)$, and essentially represents the set of states that have already been determined to be reachable. The transition relation is denoted $R(x, x')$. Both x and x' denote the boolean variables used to describe a single state. State $s1$ has a transition to $s2$ if the combination of the valuations of the variables representing $s1$ and $s2$, denoted $v1$ and $v2$, is present in the transition relation, denoted as follows: $R(v1, v2)$.

Reachability analysis is done by an iterative process that keeps on finding new reachable states until no more reachable states can be found. The difference with enumerative model checking is that as BDD operators are set operators we do not look at each state separately but instead compute the next states for all currently known states at once. This gives rise to the idea of levels: Starting with the set of initial states, the next level can be computed by computing all reachable states from the set of initial states. The result is then used in an iterative process to compute the next level which amounts to computing all reachable states from the states that have been reached until now. The computation stops when no new reachable states have been found in the last iteration, signifying that all reachable states have been found. Looking

at Figure 1.1.1 the only changed aspect is how line 5 is handled.

Going from one level to the next level can be formulated as follows:

Formula 5
$V'(x) = (\exists x(V(x) \wedge R(x, x')))[x' := x]$

$V(x)$ represents the current visited set, or in other words the set of states that have currently been found to be reachable. After applying the described operations we obtain $V'(x)$ which denotes the newly updated visited set. The formula consists of three operations that we will now discuss in detail. The first part of the formula is $V(x) \wedge R(x, x')$. Here, the AND operator is applied to the current visited set and the BDD representing the transition relation. The result of this operation is a BDD with two sets of variables, x and x'. Let us denote this result by $Result(x, x')$. As the AND operator only allows for valuations that are both in $V(x)$ and R(x,x') to appear in $Result(x, x')$, we obtain a BDD that only has transition representations of which the first set of variables (x) denotes a state that was also in $V(x)$. Therefore, all relations in the result describe transitions that can be taken by the states in $V(x)$. The second part of the formula is $\exists x(Result(x, x'))$. The $\exists$ stands for existential quantification which effectively deletes the variables described in its first argument (x), from its second argument $Result(x, x')$. So the new result after applying the $\exists$ operator is a BDD representation of all reachable next states from the set $V(x)$. Let us denote this intermediate result as $Result2(x')$. The combination of the BDD AND operator and the existential quantification is called the relational product. Even though we have now obtained the set of reachable states from $V(x)$ a last BDD operation denoted by $[x' := x]$ is needed. This operation effectively renames all variables in x' to variables in x, making sure that the result of our computations can be used in a next iteration. The final result is thus a new (updated) visited set denoted by $V'(x)$. In order to make sure that we remember all visited states (and not start looping around in our search), we should combine the new visited set with the previous visited set with the BDD AND operator.

1.2 Partitioning of the relational BDDs

The main problem of symbolic model checking is that the transition BDD and the intermediate results of BDD operations such as the relational product can get very large [16, 9]. The intermediate results of the relational product operator are often greater than the end result of the operation. Splitting the complete transition BDD into multiple transition BDDs (partitions) that together describe the complete transition system is a much used solution to this problem. In terms of storage requirements it is often the case that the combined number of BDD nodes used to describe the smaller transition BDDs is often smaller than the number of BDD nodes used to describe the original transition BDD. Computation time can also be reduced because of the existence of special relational product operators that take advantage of these smaller transition BDDs to ensure smaller intermediate results [9]. The complete transition BDD is often called the monolithic transition BDD. This section discusses the two main flavours of transition BDD partitioning: disjunctive and conjunctive partitioning.

1.2.1 Disjunctive partitioning

Disjunctive partitioning means that a disjunctive combination of multiple partitions is used to describe the monolithic transition BDD. A partition of the transition relation describes a subset of the monolithic

transition relation both in the sense that it describes part of all transitions of the model, but also in the sense that it might describe partial transitions. A partial transition is a transition that describes change for a strict subset of the total amount of model variables. The semantics of such a partial transition in the case of disjunctive partitions is that this transition may take place independently of the valuation of other model variables that are not described by this partial transition. The partial transition BDDs are combined to yield the monolithic transition BDD by applying the disjunction operation to them. The disjunctive operator makes sure that the next states determined by each of the partitions are all valid next states of the complete model. This combinations of partial transition BDDs amounts to non-deterministic choice between multiple possible next states which is the same as the interleaving semantics in a multi-process asynchronous model as described in Section 1.3.2. If partitions actually do describe all variables of the model then their combined result is the union of their successors.

Naively we could compute the relational product by first taking the disjunction over all partial transition BDDs and then using the computed monolithic transition BDD to compute the set of next states:

Formula 6
$V'(x) = (\exists x.(V(x) \wedge (R_0(x,x') \vee ... \vee R_{p-1}(x,x'))))[x' := x]$

With p the number of partitions, R_i the i-th partition, $V(x)$ the current visited set and $V'(x)$ all possible next states. A much smarter way to compute the relational product is presented next: In order to compute the relational product, the monolithic transition BDD never has to be constructed. Instead of creating the monolithic transition BDD, we can also distribute the existential quantification over the partitions [8, 9]:

Formula 7
$V'(x) = (\exists x.(V(x) \wedge R_0(x,x')) \vee ... \vee \exists x.(V(x) \wedge R_{p-1}(x,x')))[x' := x]$

1.2.2 Conjunctive partitioning

With conjunctive partitioning, a number of partitions is defined which are combined with the conjunction operator to obtain the monolithic transition BDD [3, 8]. Conjunctive partitions describe partial transitions in the sense that they contain all information of how a single variable changes. The conjunctive combination of these partial transition BDDs do not represent interleaving semantics as is the case with disjunctive partitions. Instead, the combination represents "true concurrency", as described in section 1.3.1.

When computing the relational product over the conjunctive partitions we cannot use the same trick as with the disjunctive partitions because existential quantification and logical conjunction cannot distribute [9]. The naive approach to computing the relational product is to construct the monolithic transition BDD when computing the relational product:

Formula 8
$V'(x) = (\exists x.(V(x) \wedge (R_0(x,x') \wedge ... \wedge R_{p-1}(x,x'))))[x' := x]$.

With p the number of partitions, R_i the i-th partition, $V(x)$ the current visited set and $V'(x)$ all possible next states.

Early quantification

Combining the partial transition BDDs into a monolithic transition BDD is contrary to what we set out to achieve. Early quantification is a technique which builds the relational product in a stepwise way carefully preventing the need to build the monolithic transition BDD [8, 13]. The technique is based on two observations called locality and early quantification. The locality observation states that every single partial transition BDD only depends on a small number of variables. The early quantification observation states that variables can be existentially quantified if they are not present in any of the remaining partial transition BDDs. The idea is to conjunctively combine the visited set with one of the partial transition BDDs and to existentially quantify the variables in x that are not present in any of the other partial transition BDDs. The set of variables that can be existentially quantified after conjunctively applying partition i to the intermediate results is denoted E_i. This process, of applying a partition and then using existential quantification for set E_i, is iterated for every partial transition BDD. These computations give rise to $p-1$ intermediate results denoted by $I_j(x,x')$, with j the partition last used. The last iteration should apply existential quantification on all remaining variables in x, such that the next result is $V'(x')$. After renaming we obtain $V'(x)$.

Formula 9

$I_0(x,x') = \exists E_0.(V(x) \wedge R_0(x,x'))$
$I_1(x,x') = \exists E_1.(I_0(x,x') \wedge R_1(x,x'))$
...
$I_{p-2}(x,x') = \exists E_{p-2}.(I_{p-3}(x,x') \wedge R_{p-2}(x,x'))$
$V(x') = (\exists E_{p-1}.(I_{p-2}(x,x') \wedge R_{p-1}(x,x')))[x' = x]$

The ordering in which the conjunctive partitions are used in this process is very important as it is better for both time and memory to eliminate as much variables as possible as early as possible.

1.3 Model types

Models are often described as a collection of multiple components. Such components can represent separate (but not independent) parts of a model, but they can also represent multiple independent processes. There are two different types of models with as their main difference the way that they handle parallel composition. Parallel composition is the behaviour of a system of two or more combined components. What are the transitions possible in a system that combines multiple components? Is is possible for components to independently take steps or do all components need to do a transition at the same time? The two main ways to look at parallel composition are called “true-concurrency” and “interleaving semantics” and give rise to respectively “synchronous” and “asynchronous” models.

1.3.1 Synchronous models

A synchronous model is a model that is made from a parallel composition of one or more components, which we will call modules from now on. The parallel composition of multiple modules is defined to be fully synchronous: No module may take transitions on their own, instead all modules have to take a transition at the same time. The parallel composition of a single component is the unchanged component. This type of parallel composition is called “true-concurrency” and is formally defined as follows:

Definition 10 (Parallel Composition according to True Concurrency)
Let $TS = \langle S, R, s^0 \rangle$, $TS_1 = \langle S_1, R_1, s_1^0 \rangle$ and $TS_2 = \langle S_2, R_2, s_2^0 \rangle$ be three transition systems. We define the parallel composition $TS = TS_1 \mid\mid_{sync} TS_2$, following the principle of true-concurrency, as follows:

- $S = S_1 \times S_2$
- $s^0 = s_1^0 \times s_2^0$
- $R = \{((s_1, s_2), (t_1, t_2)) \mid R_1(s_1, t_1) \wedge R_2(s_2, t_2)\}$

Synchronous models are often described with a conjunctively partitioned transition relation as the relational product operator for conjunctive partitions (described in section 1.2.2) effectively provides true-concurrency semantics.

1.3.2 Asynchronous models

Asynchronous models are made up of one of more processes. These processes are expected to behave following interleaving semantics: If a transition of a process is defined to be independent of the other processes (an internal transition) then this transition can take place separately of the other processes. Dependent, also called external, transitions communicate with transitions in other processes. All communicating transitions must happen at the same time. A process that wants to execute an external transition must wait until other processes that are defined to communicate with this transition are also ready to take the transition. External transitions therefore define communications between processes. They can be used to communicate values or to make sure that the complete system is in a certain state. An important effect of interleaving semantics is that if multiple processes have enabled internal actions, then the order in which those actions are executed in the parallel composed system is arbitrary. The order is determined by non-deterministically selecting transitions in arbitrary order. The effect on the state space is that for certain states a multitude of successors states is possible: One where process A takes an internal step, one where process B takes an internal step and so on. Determining which transitions in which processes is done by giving each transition a label. The label can be either internal denoted by τ, or external denoted by an alphabetic character. A Transition System with labels on the transitions is called a Labelled Transition System which we will define next, after which we define parallel composition works.

Definition 11 (Labelled Transition System)
A labelled transition system (LTS) is a structure $\langle S, R, s^0, L \rangle$, where S is a set of states, τ is the special internal label, L a set of labels, $R \subseteq S \times L \cup \tau \times S$ is a transition relation in which each transition is labelled with a label and $s^0 \subseteq S$ is the initial state. The set of next states of a state s, is defined to be $\{s' \in S \mid R(s, s')\}$.

Definition 12 (Parallel Composition according to Interleaving Semantics)
Let $TS_1 = \langle S_1, L_1, R_1, s_1^0 \rangle$ and $TS_2 = \langle S_2, L_1, R_2, s_2^0 \rangle$ be two labelled transition systems. We define the parallel composition $LTS = LTS_1 \mid\mid_{async} LTS_2$, following the principle of interleaving semantics, to be $LTS = \langle S_1 \times S_2, L_1 \cup L_2, R, (s_1^0, s_2^0) \rangle$, with:

$$\begin{aligned} R = &\{((s_1, s_2), a, (t_1, t_2)) \mid R_1(s_1, a, t_1) \wedge R_2(s_2, a, t_2) \wedge a \in L_1 \cap L_2\} \\ &\cup \{((s_1, a, s_2), (t_1, s_2)) \mid R_1(s_1, a, t_1) \wedge a \in L_1 \backslash L_2 \text{ or } a = \tau\} \\ &\cup \{((s_1, a, s_2), (s_1, t_2)) \mid R_2(s_2, a, t_2) \wedge a \in L_2 \backslash L_1 \text{ or } a = \tau\} \end{aligned}$$

Definition 12 describes parallel composition for only two processes in the sense that the definition does not support communications between more than two processes at once. A parallel composition between three LTS's LTS_1,LTS_2 and LTS_3 could be defined as follows: $LTS = LTS_3 \mid\mid_{async} (LTS_1 \mid\mid_{async} LTS_2)$. Depending on the semantics of a particular model checking language such multi-communication transitions are allowed or disallowed. Asynchronous models are often described with disjunctive partitions. Sets of internal transitions can be described with the use of disjunctive partitions (see Section 1.2.1). The monolithic transition BDD resulting from a disjunctive combination of the aforementioned disjunctive partitions is exactly the parallel composition following interleaving semantics of the processes described by the aforementioned disjunctive partitions.

1.4 NuSMV

NuSMV is a symbolic model checker for the SMV modelling language[1]. NuSMV is an open source project under the LGPL licence which allows full rights to use and modify NuSMV for research and commercial applications. NuSMV supports both BDD-based and SAT-based model checking **??**. In this work we will only use the BDD-based model checking approach when comparing NuSMV with LTSmin. The NuSMV model checker uses a conjunctively partitioned transition BDD for which SMV was specifically designed: The SMV language descriptions requires the user to encode variable change on a per variable basis which are easily translatable to conjunctive partitions. Even though SMV models are largely used in domains that use synchronous models it is also possible to specify asynchronous modules (a kind of asynchronous processes in an otherwise synchronous system). This function is however deprecated but can be reconstructed by the user by creating their own process scheduler (more on that later). NuSMV uses the early quantification technique described in Section 1.2.2 in order to speed up their reachability search [2].

1.5 LTSmin

LTSmin is a model checking toolset [3]. The symbolic reachability tool in the LTSmin toolset support multiple input languages for which in the most cases enumerative model checkers already exists. The languages are supported via language modules that provide a connection between the input language and the PINS interface, which is an interface that enables the LTSmin tools to query the input languages in a generalized way. Multiple input languages are supported, such as the state-based languages Promela and DVE, the two action-based process algebras μCRL and mCRL2, and the UPPAAL xml modelling language. The goal of LTSmin is to apply methods for symbolic state space exploration onto existing enumerative state generators [6, 4].

1.5.1 LTSmin PINS Interface

A much used interface between enumerative model checkers and input languages is the monolithic next-state interface [5]. This interface specifies functions that represent two requests: The request to supply the initial state(s), and the request to supply the next state(s) given a current state:

[1] http://nusmv.fbk.eu/
[2] http://nusmv.fbk.eu/NuSMV/papers/sttt_j/html/node24.html
[3] http://fmt.cs.utwente.nl/tools/ltsmin/

Definition 13 (Monolithic Next-state Interface)
A monolithic next-state interface is an interface that supports at least two functions of the following form ⟨ return type, function (arguments) ⟩:

- state set GetInitialStates();
- state set GetNextStates(state);

The LTSmin PINS interface demands from users of this interface that they support a disjunctive partitioned version of the next-state interface and that they provide a dependency matrix. PINS stands for "Interface based on a Partitioned Next-State function". The dependency matrix captures information on how transitions depend on specific parts of the system state. This extra information, apart from the next-state function gives LTSmin the information it needs to support efficient partitioned symbolic model checking. A compact explanation of PINS is as follows: A state for PINS is a vector of N slots, where a single slot can represent anything. The transition relation is split disjunctively into K groups. The N × K PINS dependency matrix then denotes which slot each group depends on [4]. Let's elaborate on this compact explanation by taking a closer look at each of the components separately: The states, the transition relation and the dependency matrix. A PINS state is a vector divided into N slots.

Definition 14 (PINS State)
A state for PINS is a vector of N slots, denoted $\langle S_1, ..., S_N \rangle$. A single slot can represent anything.

Systems often have a natural state partitioning. Normally each slot is used to represent one variable in the state space. The formal definition of the used partitioned transition system is as follows:

Definition 15 (Partitioned Transition System)
A partitioned transition system (PTS) is a structure $P = \langle \langle S_1, ..., S_N \rangle, \langle R_1, ..., R_K \rangle, \langle s_1^0, ..., s_N^0 \rangle \rangle$. The sets of elements $S_1, ..., S_N$ define the set of states $S_P = S_1 \times ... \times S_N$. The transition groups $R_i \subseteq S_P \times S_P, (1 \leq i \leq K)$ define the transition relation $R = \bigcup_{i=1}^{K} R_i$. The initial state is $s^0 := \langle s_1^0, ..., s_N^0 \rangle \in S_P$. The defined TS of P is $\langle S_P, R, s^0 \rangle$ [4].

The described transition system thus consists of:

- The set of states defined by all possible combinations of the valuations of the elements $S_1, ..., S_N$.
- The transition relation defined by a disjunctively partitioned set of partial transition relations.
- The initial state defined by the initial contents of each slot $S_1, ..., S_N$.

Note that not all states in S_P have to be reachable. S_P only denotes all possible combinations of valuations of the defined slots. Reachability analysis is needed to determine the set of reachable states. As the transition groups are indicated to define the transition relation $R = \bigcup_{i=1}^{K} R_i$, it is clear that LTSmin uses disjunctive partitioning of the transition relation. Defining the state and transition relation in this way enables the following definition of independence between transition group i and state slot j:

Definition 16 ((In)dependence between a Transition Groups and a State Slot)
Given a PTS $P = \langle \langle S_1, ..., S_N \rangle, \langle R_1, ..., R_K \rangle, \langle s_1^0, ..., s_N^0 \rangle \rangle$. Transition group i is independent of state slot j if for all $\langle s_1, ..., s_N \rangle$ and $\langle t_1, ..., t_N \rangle \in S_P$, whenever $\langle s_1, ..., s_j, ..., s_N \rangle R_i \langle t_i, ..., t_j, ..., t_N \rangle$, then

1. $s_j = t_j$ (i.e., state slot j is not modified in transition i) and

2. for all $r_j \in S_j$, we also have $\langle s_1, ..., r_j, ..., s_N \rangle R_i \langle t_1, ..., r_j, ..., t_N \rangle$. (I.e., the value of state slot j is not relevant in transition i.)

The data structure used to record the dependence or independence between state slots and transition groups is called a dependency matrix and is defined in the following way:

Definition 17 (Dependency Matrix)
A dependency matrix $D_{K \times N} = DM(P)$ for PTS P is a matrix with K rows and N columns containing $\{0, 1\}$ such that if $D_{i,j} = 0$ then group i is independent of element j.

The dependency matrix in LTSmin gives an implementation of the notion of Event Locality.

Notion 18 (Event Locality)
The notion of Event Locality refers to the fact that even though in a state several events could be enabled, each event separately affects just a small part of the state vector [5].

An important question to ask about LTSmin is the following one: "How is LTSmin able to use symbolic model checking algorithms on languages that are meant for their own enumerative model checkers?" The main obstacle for LTSmin is the creation of the symbolic transition relation. Creating the symbolic transition relation directly from the input language is impossible as enumerative input languages were not developed to support this usage contrary to the SMV language which is almost directly translatable. The naive way of building the transition relation is to query the language module for the set of next states for every possible state in S_P. This will be a costly process as the number of states in S_P is defined to be all possible combinations between the state slots. Devising an on-the-fly plan where the language module is only queried for the next states that are actually reachable is a better solution. LTSmin instead uses an even better solution of using the independence information from the dependency matrix to only query relevant combinations of the state slots. If a group of transitions is independent of slot j then LTSmin can combine the transition information of this group with the identity matrix for slot j in order to obtain all transitions for this group with the slot j unaffected in all transitions. In order to define this formally we first define the projection of the state space vector in relation to a certain transition group.

Definition 19 (Projection)
For any transition group $1 \leq i \leq K$, we define π_i as the projection $\pi_i : S \to \Pi_{\{1 \leq j \leq N | D_{i,j} = 1\}} S_j$

The projection of state vector S to a subset of the state vector S's slots is a combination of the original slots for which the condition holds that the dependency matrix hold value "1". The effect of this definition of the relational transition BDD is that the LTSmin language modules are only queried for the combinations of the state slots represented in $R_g(\pi_g(x), \pi_g(x'))$. All combinations with the independent variables are never queried: the independent variables are inserted into the partial transition BDD as non changing variables. The total relational BDD can then be described in the following way:

Formula 20
$R(x, x') = \bigvee_{g=1}^{K} (R_g(\pi_g(x), \pi_g(x')) \wedge \bigwedge_{i \notin I_g} [x_i = x'_i])$[5]

Where g denotes a transition group, I_g denotes set of state slots depended on group g, $\pi_g(x) = (x_j)_{j \in I_g}$ denotes the projection to a transition group and $[x_i = x'_i]$ denotes the BDD representing the identity matrix for all independent variables. Something similar is done in [3]. Disjunctive partitions are kept small by not recording the variables that are independent, and the relational product operator is adjusted to be able to work with these new BDDs that do not contain information on all variables. By keeping the variables that do not change out of the BDDs lots of memory space is saved.

The PINS version of the next-state interface which exposes dependency information takes the following form:

- int GetStateLength();
- int GetGroupCount();
- int list GetGroupInfluenced(int group);
- state GetInitialState();
- state set GetNextStates(state src, int group);

The function GetStateLength enables obtaining information about the state partitioning. The length is the number of slots that are defined. GetGroupCount gives the amount of disjunctive transition partitions. GetGroupInfluenced is the way to communicate about the contents of the dependency matrix. The answer is a list of integers that show which state slots are dependent on the given group. GetInitialState is trivial. The GetNextStates function is now adapted to incorporate the option of generating next states with respect to the set of state slots defined by the given group.

1.5.2 LTSmin language modules

A LTSmin language modules is a body of (glue) code that enables LTSmin to use an input language through the PINS interface. Such a language module usually attempts to link to the existing model checking tool for the target input language in order to reuse their parser and possibly more existing functionalities. Reusing code by linking to the input languages existing toolset ensures that creating a language module is manageable in time and it also prevents problems in the future if the input language is changed. If LTSmin would contain a complete parser for all input languages supported then those would have to be updated with all updates to the input languages. Linking to existing tools is done by loading libraries containing the code of the already existing tool in question and by implementing the PINS interface by using those loaded libraries.

1.5.3 LTSmin and NuSMV compared

The two main differences between LTSmin and NuSMV are:

- LTSmin supports multiple input languages versus NuSMV that only supports the SMV language.
- LTSmin uses disjunctive partitioning of the translation BDD versus NuSMV which uses conjunctive partitioning of the transition BDD.
- LTSmin has to learn the transition BDD via the PINS interface versus NuSMV whose input language is directly translatable to a transition BDD.

Both NuSMV and LTSmin support asynchronous and synchronous models, even though asynchronous have been deprecated for NuSMV.

Chapter 2

Automated Translations

This chapter describes, formalizes and provides correctness proofs for the implemented automatic translations. To that end, the used languages are clearly defined, the translations described and proofs presented. First we will give a preview and examples of the different types of models and the translations between them. Afterwards we will start with describing and formalising the SMV language which we call the SMV-input language. Then, we will explain the SMV to SMV-flat translation which flattens an SMV-input model to an SMV-flat model. The SMV-input model may contain multiple synchronous and or asynchronous modules whereas the SMV-flat model may contain only one synchronous module. The flattening translation translates from SMV-input models to SMV-flat models while preserving the model's behaviour.

Once the SMV to SMV-flat translation has been described and proven correct a description follows of the translations that actually translate from SMV-flat to mCRL2. To that end, we describe and formalize mCRL2 and prove two translations that translate from SMV-flat towards mCRL2.

2.1 Preview

This section provides a short explanation and examples of the SMV to SMV-flat, the SMV-flat to mCRL2 All-in-1 and the SMV-flat to mCRL2 1-by-1 translations. The aim of this Section is to provide an overview picture of the languages involved and the created translations before going into formalisations and proofs.

2.1.1 SMV to SMV-flat translation

Let's first take a look at the SMV to SMV-flat translation. This translation has as its input a normal SMV model in textual form. We call this textual model an SMV-input model. The result is a data-structure called SMV-flat that is directly translatable to a textual SMV model. The main difference between the SMV-input and the SMV-flat model is that SMV-input models may have multiple (possible asynchronous) modules whereas the SMV-flat model has only a single synchronous module. Both models however still express the same behaviour. The transformation done by the translation is called "flattening". The resulting SMV-flat model contains much less SMV language features and is therefore easier to translate to an mCRL2 model.

While compressing all modules into the single "main" module, we have to consider the two types of modules that the SMV language supports. Synchronous modules adhere to the concept of true concurrency (Def. 10) together with all other synchronous modules and the "main" module. Asynchronous modules adhere to the principle of interleaving semantics (Def. 12) without the option to synchronize with other modules in any way. Expanding a given state into it's next states therefore works as follows: First a module is chosen for execution. This may be any of the asynchronous modules, or the combination of the "main" module and all synchronous modules. Afterwards, only the statements (defining how the state variables change) in the chosen module(s) are executed. From now on we will act as if the combination of the "main" module and all synchronous modules is a single asynchronous module.

The resulting single "main" module in the SMV-flat model must exert the same behaviour which is ensured with the introduction of a scheduler variable that non-deterministically determines which asynchronous module is to be executed next. The scheduler variable is of the enumeration type with a single symbolic constant for each possible asynchronous module. The expressions in all modules that influence the state variables are extended with the condition that they may only be executed if their module is scheduled for execution by the scheduler variable. If not, then their influenced variable stays the same.

Let's look at an improvised example to get a clearer picture of the proposed translation. The example consists of four modules, a sender, a receiver, a communication channel and a "main" module. The sender and receiver modules are asynchronous modules as is reflected by the use of the "process" keyword when they are instantiated in the "main" module. The channel module is a synchronous module as reflected by the absence of the "process" keyword. In the example, the sender produces a message and puts into the channel. The receiver attempts to read the channel effectively clearing it of the message. The channel itself has a data variable which represents the contents of the channel. The channel is also able to non-deterministically put the channel into an error state but can only do this if the channel is currently empty. The main module is used to tie all other modules together, giving the sender and receiver modules both access to the channel.

```
1  MODULE Channel
2  VAR
3    data : {EMPTY, FULL, ERROR};
4
5  ASSIGN
6    init(data) := EMPTY;
7    next(data) := case
8      data = EMPTY : {EMPTY,ERROR};
9      TRUE : data;
10   esac;
11
12 MODULE main
13 VAR
14   channel : Channel;
15   send : process Sender(channel);
16   receiv : process Receiver(channel);
```

```
1  MODULE Sender(chan)
2  ASSIGN
3    next(chan.data) := case
4      chan.data = EMPTY : FULL;
5      TRUE : chan.data;
6    esac;
7
8  MODULE Receiver(chan)
9  ASSIGN
10   next(chan.data) := case
11     chan.data = FULL : EMPTY;
12     TRUE : chan.data;
13   esac;
```

The channel is initialized to EMPTY and both asynchronous modules Sender and Receiver have access to the channel's data variable via their module parameter chan. Execution of this model happens by selecting either the Sender, the Receiver or the combination of the main and Channel module for execution,

after which their next declarations are applied to the state variables. All unmentioned state variables stay the same. The following example does just that, while using only the single "main" module.

```
1  MODULE main
2  VAR
3    channel_data : {EMPTY, FULL, ERROR};
4    scheduler : {sched_main, sched_send, sched_receiv};
5
6  ASSIGN
7    init(channel_data) := {EMPTY};
8    next(channel_data) := case
9      scheduler = sched_main : case
10       channel_data = EMPTY : {EMPTY,ERROR};
11       TRUE : channel_data;
12     esac;
13     scheduler = sched_send : case
14       channel_data = EMPTY : FULL;
15       TRUE : channel_data;
16     esac;
17     scheduler = sched_receiv : case
18       channel_data = FULL : EMPTY;
19       TRUE : channel_data;
20     esac;
21   esac;
22
23   init(scheduler) := {sched_main, sched_send, sched_receiv};
24   next(scheduler) := {sched_main, sched_send, sched_receiv};
```

Note that the SMV-input model has 3 states, whereas the SMV-flat model has 9. This is due to the fact that the non-deterministic scheduler variable is added to the state vector which effectively generates multiple versions of the same SMV-input states: one version for each possible scheduler choice.

2.1.2 The SMV-flat to mCRL2 "All-in-1" translation

mCRL2 models consist of processes, which in turn consists of summands. Using only a specific subset of the mCRL2 language, we are able to describe the changes in the state variables as expressed in the SMV-flat model. For our translations we only create a single mCRL2 process, with summands that have an optional condition, an unused action and a process-call expression. The process-call expression is responsible for the change in data variables from one state to another. The optional condition is used in order to determine if the change in data variables expressed by this summand is enabled for the current state. Actions actually label the transitions of an mCRL2 model but as the SMV language does not use the concept of actions we actually have no need of mCRL2 actions as well. The action is however non-optional in the mCRL2 language and therefore the translation sets it to the improvised action called "noAction".

Reachability analysis of an arbitrary mCRL2 model works by expanding already found states starting with the unique initial state. As we always translate to models that only have a single mCRL2 process we only take that process's summands into consideration. Expansion of a state then works by non-deterministically selecting an enabled summand. The enabledness of a summand is determined by the validity of the condition in the current state. The process-call expressions within the enabled summands are then used to determine the next states as they describe next state valuations for all state variables (non described variables remain the same).

The SMV language supports non-determinism when specifying a variables valuation in the next state. mCRL2 describes non-determinism by having a summand for each non-deterministic option. We therefore translate each non-deterministic choice in the SMV-flat model to a summand in the mCRL2 model. The All-in-1 translation generates a summand for each combination of possible non-deterministic choices available in the SMV-flat model. In this way the same non-deterministic choices can be made in both the mCRL2 and the original SMV-flat model.

Some SMV models may have multiple initial states. In our case, where we first use the SMV to SMV-flat translation this is even more likely as that translation introduces the non-deterministic scheduler variable that also has a non-deterministic initial valuation (creating multiple initial states). As mCRL2 models only support the presence of a single unique initial state we need to introduce a new initial state within the mCRL2 model that first generates the SMV-flat initial states before continuing with the generation of the normal next states.

In order to do this we add a special initial flag variable to the state vector of the mCRL2 model that indicates whether a state is the unique mCRL2 initial state. In order to generate the set of SMV initial states we then add summands to the mCRL2 models that have as their condition that the initial flag must be true, and as their process-call expression state variable changes that generate the states in the SMV initial states set. The process-call expression must also set the initial flag to false.

The originally created summands that generate next states from all already generated states must be extended with the condition that the initial flag must be false. Let's give a translation of the example presented in Section 2.1.1 in order to see how the "All-in-1" translation works.

```
1  sort Enum0: {EMPTY, FULL, ERROR};
2  sort Enum1: {s_main, s_send, s_receiv};
3
4  proc P(c_d: EnumType0, s: EnumType1, IF: Bool){
5  (IF = True) -> (noAction . P(c_d = EMPTY, s = s_main, IF = False))
6  + (IF = True) -> (noAction . P(c_d = EMPTY, s = s_send, IF = False))
7  + (IF = True) -> (noAction . P(c_d = EMPTY, s = s_receiv, IF = False))
8  + (IF = False) -> (noAction . P(c_d = if(s = s_main,if(c_d = EMPTY, EMPTY,if(True,c_d,c_d)))
9          , s = s_main, IF = False))
10 + (IF = False) -> (noAction . P(c_d = if(s = s_main,if(c_d = EMPTY, ERROR,if(True,c_d,c_d)))
11         , s = s_main, IF = False))
12 + (IF = False) -> (noAction . P(c_d = if(s = s_main,if(c_d = EMPTY, EMPTY,if(True,c_d,c_d)))
13         , s = s_send, IF = False))
14 + (IF = False) -> (noAction . P(c_d = if(s = s_main,if(c_d = EMPTY, ERROR,if(True,c_d,c_d)))
15         , s = s_send, IF = False))
16 + (IF = False) -> (noAction . P(c_d = if(s = s_main,if(c_d = EMPTY, EMPTY,if(True,c_d,c_d)))
17         , s = s_receiv, IF = False))
18 + (IF = False) -> (noAction . P(c_d = if(s = s_main,if(c_d = EMPTY, ERROR,if(True,c_d,c_d)))
19         , s = s_receiv, IF = False))
20 }
21
22 init allow(noAction,P(EMPTY,sched_main,True);
```

As can be seen in the example on the previous page all case-esac expressions of the SMV-flat model have been translated into if-then-else expressions from mCRL2 (denoted if(condition-expression,then-expression,else-expression)). The reachable state spaces of the SMV-flat and mCRL2 "All-in-1" translations are exactly the same with the exception of the special mCRL2 initial state. The difference in the reachable state space

is pictorially described in Figure 2.1.2.

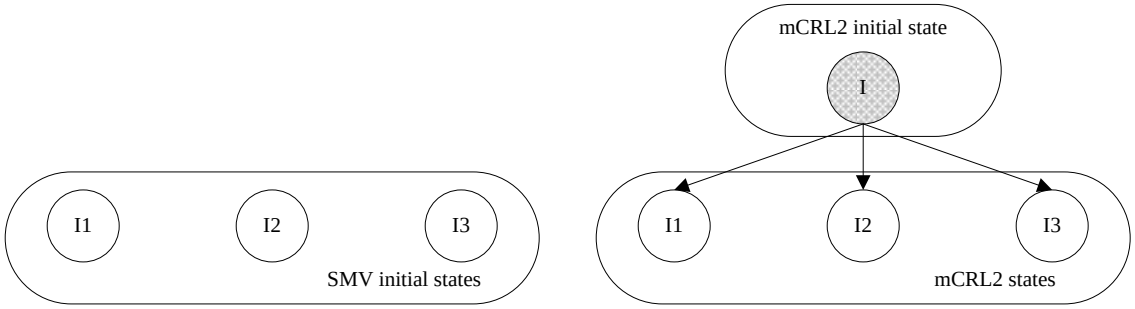


Figure 2.1: The SMV-flat initial states versus the single mCRL2 initial state

The unique mCRL2 initial state is greyed out because it is not really a part of the comparable reachable state space as it is actually used as a computation step to generate the SMV initial state set.

2.1.3 The SMV-flat to mCRL2 "1-by-1" translation

All-in-1 models have a summand for each combination of non-deterministic choices for all state variables. The process-calls in a All-in-1 model change all the state variables at the same time. As explained in Section 1.5.1, LTSmin benefits from it's dependency matrix that tells LTSmin which transition groups depend on which variables. LTSmin creates the dependency matrix from an mCRL2 model by creating a "dependency group" for each summand, for which the group depends on the variables mentioned in the summand's condition and process-call. If we thus change all variables in every process-call then that means that according to LTSmin all created groups depend on all variables for all All-in-1 translated models. As LTSmin cannot construct a beneficial dependency matrix this probably prevents LTSmin from executing an efficient reachability search.

The 1-by-1 translation attempts to inspire LTSmin to create a better dependency matrix by only changing a single process variable in each created summand. In order to only change a single variable per summand and still have a model that behaves in the same way as the SMV-flat model we need to have some kind of counter variable that governs the correct execution of which variable is to be changed next. The counter variable is of integer type and is initialised to zero, and ranges to the number of model variables plus 1 and is then reset to zero. Each variable is assigned a unique ID starting with zero which indicates when they should be changed according to the value of the counter variable. For each variable we therefore create a summand, which has as its condition that the counter variable has the correct ID and as its process-call an expression that changes that single variable to its next state valuation according to the SMV-flat model. We also increment the counter variable by 1 in order to make sure that in the next state of the 1-by-1 model the next variable will be evaluated. Once all variables have been evaluated we have successfully simulated a single transition of the SMV-flat model.

Non-determinism in the SMV-flat model can be build in into this 1-by-1 model in an efficient way: If an SMV variable has a non-deterministic choice as its valuation, then we create multiple summands that express all possible non-deterministic choices for that variable. All summands get the same condition

which expresses that the counter must be equivalent to the variable's ID and as such all the summands that enumerate the possible non-deterministic choices are enabled at the same time. If the variable has multiple non-deterministic choices then we once again create a summand for each combination of possible options of those non-deterministic choices. This however blows up much less than in the case of the All-in-1 translation as we only take into account the non-deterministic choices for a single variable. The example below shows the creation of multiple summands for the scheduler variable.

The 1-by-1 translation also has the previously discussed solution for multiple initial states built in with the initial flag: The unique mCRL2 initial state is first used to construct all SMV-flat initial states after which we continue with the generation of the rest of the state space. Another important feature of the 1-by-1 model is that all model variables have to be recorded twice: When computing the next state valuation of a variable we might need the current state valuations of any of the other variables. These must still be available even if their next state valuations have already been computed. We therefore need two variables for each variable, one that contains the current state valuation and one that contains the possibly already computed next state valuation. After computing all variable's next state valuations we have an extra summand that copies the next state valuations to the variables that record the current state valuations. In our example we call the current state variable by its original name, and for the next state variable we add "_e".

```
1  sort Enum0: {EMPTY, FULL, ERROR};
2  sort Enum1: {s_main, s_send, s_receiv};
3
4  proc P(c_d: Enum0, c_d_e: Enum0, s: Enum1, s_e: Enum1, IF: Bool, PC: Int){
5  (IF = True & PC = 0) -> (noAction . P(c_d_e = EMPTY, PC = 1)
6  + (IF = True & PC = 1) -> (noAction . P(s_e = s_main, PC = 2)
7  + (IF = True & PC = 1) -> (noAction . P(s_e = s_send, PC = 2)
8  + (IF = True & PC = 1) -> (noAction . P(s_e = s_receiv, PC = 2)
9  + (IF = False & PC = 0) -> (noAction . P(
10         c_d_e = if(s = s_main,if(c_d = EMPTY, EMPTY,if(True,c_d,c_d))), PC = 1))
11 + (IF = False & PC = 0) -> (noAction . P(
12         c_d_e = if(s = s_main,if(c_d = EMPTY, ERROR,if(True,c_d,c_d))), PC = 1))
13 + (IF = False & PC = 1) -> (noAction . P(s_e = s_main, PC = 2))
14 + (IF = False & PC = 1) -> (noAction . P(s_e = s_send, PC = 2))
15 + (IF = False & PC = 1) -> (noAction . P(s_e = s_receiv, PC = 2))
16 + (PC = 2) -> (noAction . P(c_d = c_d_e, s = s_e, IF = False, PC = 0)
17 }
18
19 init allow(noAction,P(EMPTY,EMPTY,sched_main,sched_main,True,True,0);
```

As mCRL2 models are interpreted by creating a state and a transition for each enabled process-call we have effectively simulated one SMV-flat transition with multiple mCRL2 transitions (with in between states). To be more precisely: Say we have 3 variables, then we need 3 transitions to change those variables and we also need a transition to copy the newly computed next state valuations to the variables that track the current state valuations. We therefore need the number of model variables plus one transitions to simulate a single SMV-flat transition. When also taking account non-determinism this calculation becomes even more involved as depicted in Figure 2.2.

Figure 2.2 shows a comparison between a state and its next states in the SMV-flat model and in the translated mCRL2 1-by-1 model. This is a comparison for an SMV-flat model with 3 variables for which variable 2 has a non-deterministic choice with 3 choices. The numbers in parenthesis on the right represent the value of the counter variable. The greyed out states are in-between computation state in which only

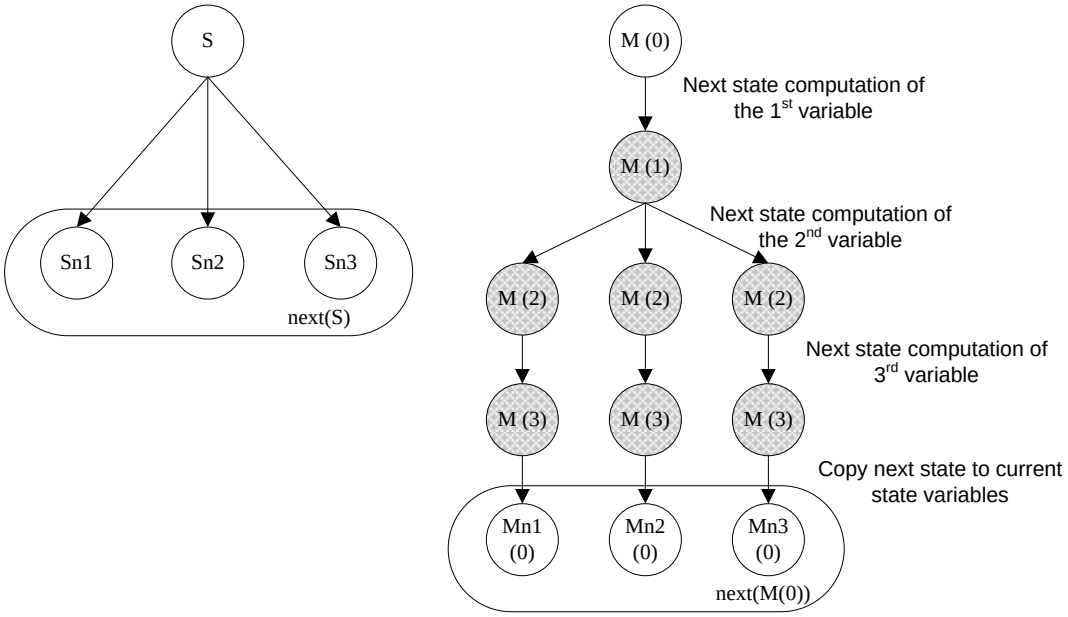


Figure 2.2: A single state and its next states from an SMV model with 3 variables, non-determinism in variable 2 with 3 options (left) and from the translated mCRL2 1-by-1 model (right)

part of the variable's next state valuations have been computed. In a comparison of the state space we therefore do not want to consider those states. Reachability of non-computational states should however be equivalent in both models.

2.2 The SMV language

In this section we will formalise the SMV language. We will only describe the parts of the language that are supported by our translations. Missing parts of the language in this description are thus not supported, of which the most important are the INIT and TRANS sections. All terms used in the definitions that are described by other definitions are printed in bold text.

Definition 21 (SMV model)
An SMV model is a collection of one or more modules. A special module with specification name "main" must be available.

Definition 22 (SMV module)
An SMV module is a collection of the following items:

1. A module specification name described by a **variable identifier**.
2. A number of parameters, each of which are symbolically named by a **variable identifier**.
3. A number of VAR sections that contain zero or more **variable declaration**s.
4. A number of ASSIGN sections that contain zero or more **init declaration**s and or **next declaration**s.
5. A number of DEFINE sections that contain zero or more **define declaration**s.
6. A number of IVAR sections that contain zero or more **variable declaration**s.
7. A number of CONSTANTS sections that contain a single **constants declaration**.

Definition 23 (SMV variable declaration)
An SMV variable declaration is a description of a single variable containing of a **variable identifier** and a **variable type**. Variable names must be unique within the same module and may not be declared multiple times within the same module. Variables with the module-type may only be defined in the special "main" module.

Definition 24 (SMV variable type)
An SMV type is a description of a variable's type, being either "boolean", "enumeration type", "integer range", "asynchronous module" or "synchronous module". The asynchronous and synchronous module types are special types that declare the initialization of a new module, its module specification type and its parameter arguments. It is allowed to give any expression as argument. Arguments can therefore be used to tie modules together by giving them each others variables as arguments to their parameters. The module types contain of the following information: The optional keyword "process" signifying this module is of the asynchronous module type, or the absence of the keyword "process" signifying that the module is of synchronous module type, the module specification name and the arguments to the specification's parameters. The module specification type may not be the special "main" module.

Definition 25 (SMV variable identifier)
An SMV variable identifier is a string that does not contain a dot ".", and does not begin with a number, but that otherwise may consist of A-Z, a-z, 0-9 and the special symbols "$","#", "_" and "-".

Definition 26 (SMV complex identifier)
An SMV complex identifier is an SMV variable identifier that may actually contain one or more dot "."

symbols. All two pairs of strings directly separated by dot symbols signify on the left of the dot the name of the variable that has a module type in which the variable as named on the right side of the dot is declared.

Definition 27 (SMV define declaration)
An SMV define declaration contains of a **variable identifier** and an **expression**. A define declaration defines a macro, or also called a function that relates the name of the function (the variable identifier) to how it is calculated (the expression).

Definition 28 (SMV init declaration)
An SMV init declaration consists of a **variable identifier** and an **expression**. The init declaration signifies that the variable described by the variable identifier must be initialized according to the valuation of the expression. Each declared variable in a **variable declaration** may only be initialized by a single init declaration. Each variable defined in the SMV model must have an init declaration associated with it, the absence of such an init declaration implicitly declares the init declaration to be the non-deterministic choice between all of the variable's possible values. The expression of this init declaration may not make use of the case-esac or next expression.

Definition 29 (SMV next declaration)
An SMV next declaration consists of a **variable identifier** and an **expression**. The next declaration signifies that the variable described by the variable identifier changes its valuation in the next state according to the expression. It is allowed to have multiple next declarations that convey information about the same variable as long as they are declared in different asynchronous modules. The combination of the synchronous modules together with the "main" module also counts as a single asynchronous module. Each variable defined in the SMV model must have a next declaration associated with it, the absence of such a next declaration implicitly declares the next declaration to be the non-deterministic choice between all of the variable's possible values.

Definition 30 (SMV constants declaration)
An SMV constants definition declares a list of **symbolic constant**s that should be added to the available symbolic constants. The constants declaration is used in combination with define variables that do not have a declaration type and therefore cannot in that way declare their enumeration type.

Definition 31 (SMV symbolic constant)
An SMV symbolic constant is a string that abides by the rules of the **SMV variable identifier** which can be used as an enumeration symbol.

Definition 32 (SMV integer constant)
An SMV integer constant is an integer number that can be used as a terminal symbol in **expression**s.

Definition 33 (SMV expression)
An SMV expression is a Directed Acyclic Graph with several vertices, labelled with expression operators or terminals. Each vertex can have a certain number of expressions as its children (denoted #). An expression

may begin with any of the available vertices. The available operators vertices are described by the table 2.1. The "any" type stands for the three normal types: boolean, integer (range) and enumeration type.

Table 2.1: SMV expression operators

Operator	#	Result Type	Child type	Description
(..)	1	any	any	precedence
! ..	1	boolean	boolean	logical NOT
next(..)	1	any	variable identifier	next state valuation*
- ..	1	integer	integer	unary minus
.. &, | ..	2	boolean	boolean	logical AND, logical OR
.. =, != ..	2	boolean	any	equality, inequality
.. <, >, <=, >= ..	2	boolean	integer	less, greater, .. or equal, .. or equal
.. +,- ..	2	integer	integer	addition, subtraction
.. *,/,mod ..	2	integer	integer	multiplication, division, remainder
.. in ..	2	boolean	integer, enumeration	set inclusion
.. union ..	2	set of child type	integer, enumeration	the union of two sets
.. ? .. : ..	3	any	**	if-then-else
case .. esac	*	any	***	case-esac***
{ .. }	*	any	any	non-deterministic choice
{ .. }	*	list of any	any, but all the same	In a set context: creation of a set

* The next valuation expression evaluates to the valuation of the pointed to variable in the next state.
** The type of the children of the if-then-else expressions differs per child: The first child must have a boolean type, and the other two may have any type.
*** The case-esac expression consists of pairs of expressions. Each pair consists of a condition **expression** and a valuation **expression**. The condition expression must evaluate to a boolean type and determines if the valuation should be used. A case-esac expression with multiple such pairs works by first analysing the first pair. If the condition of the first pair evaluates to true then the case-esac expression evaluates to the valuation of the first pair. If not then the second pair is evaluated and so forth. It is considered to be a fault on the model creator's side if there is no pair for which the condition evaluates to true when a case-esac expression is evaluated.

Terminals are either **symbolic constant**s, **integer constant**s, or **variable identifier**s that point to **variable declaration**s and or **define declaration**s. These terminals have respectively the types enumeration, integer and the type of the pointed-to variable or define declaration.

2.2.1 Unsupported parts of the SMV language

The following parts of the SMV language are not supported (or have not been looked at) in this project: word types, array types, bit shift operators, XOR, XNOR, implication (->), equivalence (<->), count, word conversion and modification functions such as word1, bool, toint, signed, unsigned, extend, resize and word concatenation (::). It is also important to note that the sections INIT and TRANS are not supported. Models with a TRANS section are not usable with the translations used in this project as they define the next state of variables in a totally different way than is done in the ASSIGN section. The sections INVAR, CTL (SPEC), LTLSPEC, PSLSPEC, FROZENVAR, ISA, COMPUTE are also not supported

and automatically removed by the translation tool. Those sections denote all kinds of specifications which have not been attempted to be translated as this project is solely about the reachability problem. The FAIRNESS section influences the reachable state space results and is as such also removed from the SMV model before determining the model's execution time on NuSMV (as is the same for all other removed specifications).

2.3 SMV to SMV-flat formalisation

SMV to SMV-flat is a translation that has as its input SMV models with one or multiple modules, and as its output an equivalent model that only consists of a single "main" module. The interesting part about this translation is how asynchronous modules are handled: A scheduler and conditions on next expressions must be added in order to govern the asynchronous execution semantics of the original model. The availability of the SMV to SMV-flat translation simplifies the translations from the resulting SMV-flat model to mCRL2 as those translations do not have to cope with the semantics of possibly present asynchronous modules.

The SMV to SMV-flat translation consists of two translations, one that translates from SMV-input to SMV-intermediate and one that translates from SMV-intermediate to SMV-flat. The SMV-intermediate model is a data-structure that helps to organise the information in the SMV-input model differently as to simplify the translation from SMV-intermediate to SMV-flat (in both theory and actual implementation). The SMV-input and SMV-flat models are translatable to an actual textual SMV model, the SMV-intermediate model is not.

In order to formally describe and prove the SMV to SMV-flat translation we first will formally describe the SMV-input and SMV-intermediate models. Then we will describe the translation algorithms from SMV-input to SMV-intermediate, after which we describe the SMV-flat model and the translation from SMV-intermediate to SMV-flat. Afterwards we will take a look at the formal semantics of SMV in order to determine how an SMV model induces a Transition System. Lastly we will use all described model types, translations and formal semantics in order to prove that the Transition System induced by an SMV-input model is Equivalence-class Bisimilar (Def. 48) to the Transition System induced by a translated SMV-flat model. The desired proof is pictorially described in Figure 2.3.

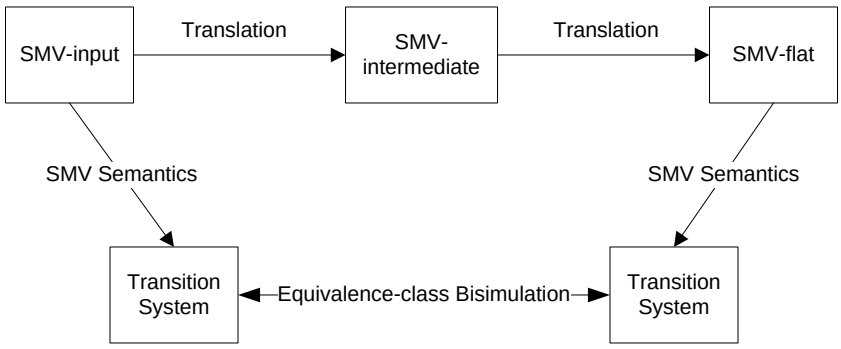


Figure 2.3: The proof obligation for the SMV to SMV-flat translation depicted.

Firstly we define the SMV-input model that is supported by the SMV to SMV-flat translation.

Definition 34 (SMV-input)
An SMV-input model is seen as a Directed Acyclic Graph containing the aspects as described in Definition 21.

As an intermediate step we define the SMV-intermediate data-structure that essentially contains the same data as the SMV-input model defined in Definition 34, but which removes the need for module parameters and next declarations that are defined in one module, but that influence variables declared in another module.

Definition 35 (SMV-intermediate)
An SMV-intermediate data-structure is a 5-tuple: ⟨Ms,Vs,Ds,Is,Xs⟩. Ms is a set of modules. Vs is a list of variables, Ds a set of define macros, Is a set of init expressions that describe the initial value of a variable and Xs is a set of next expressions that describe how a variable changes. A module denoted M is described by a 2-tuple ⟨N,B⟩, with N its name and B a boolean variable which is TRUE when the module is an asynchronous process and FALSE if not. A variable denoted V is described by a 3-tuple consisting of ⟨M,N,T⟩. With M the name of the module typed variable which instantiates the module specification that the variable was declared in, N the name of the variable and T the type of the variable. T may be any type except the module-type. A define macro denoted D is described by a 3-tuple consisting of ⟨M,N,E⟩, with E the expression that defines the valuation of the define macro. An init expression denoted I is described by ⟨M,V,E⟩, with V the state variable and E the expression of the initial value. A next expression denoted X consists of a 3-tuple: ⟨M,V,E⟩, with V a state variable and E the expression that defines the next state valuation of variable V. All expressions denoted E are expressions such as defined in definition 33 with the exception that references to module parameters have been substituted with their respective arguments and that next expressions have been substituted for the actual next expression of the variable they point to. Init declarations denoted I and next declarations denoted X follow the same syntactic rules as the **init declaration** (Def. 28) and the **next declaration** (Def. 29). Modules denoted M in set Ms are only actual modules (not module specifications). This means that a module M has the name of the variable in which it was declared. The variable declaring the module M has a module type in the original SMV-input model and is not represented in set Vs. If a module specification is not mentioned in the module-type of any variable then it is not represented in set Ms, and if a module specification is mentioned multiple times then it can also be found in set Ms multiple times. The same holds for variables, define macros, init and next declarations (their module must exist in Ms).

The translation from SMV-input to SMV-intermediate is formally described next together with the substitution function used by the translation.

Definition 36 (Expression substitution function)
Function $subst : expression \rightarrow expression$ is a function which substitutes references to module parameters with their respective arguments and substitutes next expressions for the actual next expression of the variable they point to.

Definition 37 (Translation from SMV-input to SMV-intermediate)
The translation from an SMV-input model M to an SMV-intermediate model $M_i = \langle Ms_i, Vs_i, Ds_i, Is_i, Xs_i \rangle$ is defined as follows:

- $Ms_i = \{\langle N_i, \text{TRUE} \rangle \mid \exists$ **variable declaration** VD $\in M$. VD's **type** = asynchronous module and N_i = VD's **variable identifier**$\} \cup \{\langle N_i, \text{FALSE} \rangle \mid \exists$ **variable declaration** VD $\in M$. VD's **type** = synchronous module and N_i = VD's **variable identifier**$\}$
- $Vs_i = \{\langle M_i, N_i, T_i \rangle \mid M_i \in Ms_i$ and $\exists$ **variable declaration** denoted VD $\in M$. **type** $\neq$ a module type, VD is described in the module of which M_i is an instantiation and T_i equals the type of VD$\}$
- $Ds_i = \{\langle M_i, N_i, \mathit{subst}(E_i) \rangle \mid M_i \in Ms_i$ and $\exists$ **define declaration** denoted DD $\in M$. DD is described in the module of which M_i is an instantiation, N_i = DD's **variable identifier** and E_i is the **expression** mentioned in DD$\}$
- $Is_i = \{\langle M_i, V_i, \mathit{subst}(E_i) \rangle \mid M_i \in Ms_i$ and $\exists$ **init declaration** denoted ID $\in M$. ID is described in the module of which M_i is an instantiation, $V_i \in Vs_i$, V_i = the variable pointed to by ID's **variable identifier** and E_i is the **expression** mentioned in ID$\}$
- $Xs_i = \{\langle M_i, V_i, \mathit{subst}(E_i) \rangle \mid M_i \in Ms_i$ and $\exists$ **next declaration** denoted XD $\in M$. XD is described in the module of which M_i is an instantiation, $V_i \in Vs_i$, V_i = the variable pointed to by XD's **variable identifier** and E_i is the **expression** mentioned in XD$\}$

One important feature of the SMV-input to SMV-intermediate translation is that it traces pointers to variables back to their original variable declaration and replaces those pointers with their pointed to variables. If a next declaration pointed to a variable that was not declared in the same module but instead in one of that module's parameters then they are now substituted for their actual variable declarations effectively removing the need for the parameters. We will now take a look at the exact subset of SMV that is used in an SMV-flat model

Definition 38 (SMV-flat)
An SMV-flat model consists of a single module called "main". The model is described by the 4-tuple: $\langle Vsf, Dsf, Isf, Xsf \rangle$. Vsf is a set of state variables, described by a 2-tuple ⟨Nf,Tf⟩, with Nf the name and Tf the **type** of the variable. Tf may be any type except the two module types. Dsf is a set of define macros described by a 2-tuple ⟨Nf,Ef⟩ with Nf the name and Ef the expression defining the define macro's valuation. Isf is a set of init definitions that describe the initial value of a variable with a 2-tuple ⟨Vf,Ef⟩. Xsf is a set of next definitions that consist of 2-tuples ⟨Vf,Ef⟩ that define the variables change. All expressions denoted Ef are expressions such as defined in definition 33. Init expressions denoted If and next expressions denoted Xf follow the same syntactic rules as the **init declaration** (Def. 28) and the **next declaration** (Def. 29).

In order to translate from SMV-intermediate to SMV-flat we need to define a renaming function for the state variable and define macro names. This is needed in order to guarantee unique names for the state and define macro variable in the single "main" module. We also define the scheduler variable that is used to simulate the execution of multiple asynchronous modules within the confines of a single synchronous module. Afterwards we describe the translation that translates from the SMV-intermediate data-structure to the SMV-flat model.

Definition 39 (Renaming function)
The renaming function denoted *rename* is a bijective function defined by:

$$rename(\langle module, name, ... \rangle) = \langle module\_name, ... \rangle$$

Its inverse is defined by:

$$rename^{-1}(\langle module\_name, ... \rangle) = \langle module, name, ... \rangle$$

The *rename* function operates on any tuple of size 2 or larger where the first two elements are strings. The resulting tuple is of size 1 smaller than the input tuple. The inverse function accepts any tuple of size greater than 1 for which the first element must be a string and must contain an underscore. The resulting tuple is of size one bigger than the input tuple.

Definition 40 (Scheduler variable)
The scheduler variable denoted Sc simulates the asynchronous execution of multiple asynchronous modules within a single synchronous module. The scheduler variable is uniquely named and has the enumeration type. Sc's enumeration type has a symbolic constant for each of the asynchronous modules found in the SMV-intermediate model. Assuming the existence of N such modules the symbolic constants are denoted Sc_1, Sc_2, ... Sc_N. The enumeration type also contains a symbolic constant for the combination of the main process plus remaining synchronous modules denoted Sc_m. The scheduler variable's init declaration (denoted Sc_i) and next declaration (denoted Sc_x) is a state that Sc may non-deterministically change to Sc_1, Sc_2, ... Sc_N and Sc_m. The non any of the defined symbolic constants at any point in the execution of the model.

Definition 41 (Translation from SMV-intermediate to SMV-flat)
The translation from an SMV-intermediate model $M_i = \langle Ms_i, Vs_i, Ds_i, Is_i, Xs_i \rangle$ to an SMV-flat model $M_f = \langle Vs_f, Ds_f, Is_f, Xs_f \rangle$ is defined as follows:

- $Vs_f = \{rename(V_i) \mid V_i \in Vs_i\} \cup Sc$ (Def. 40).
- $Ds_f = \{rename(D_i) \mid D_i \in Ds_i\}$.
- $Is_f = \{rename(I_i) \mid I_i \in Is_i\} \cup Sc_i$.
- $Xs_f = \{rename(\langle M_i, N_i, E_i \rangle) \mid \exists V_i = \langle M_i, N_i \rangle \in Vs_i$. E_i is a case-esac expression with a condition valuation pair called CVP for each $X_i \in Xs_i$ for which the variable equals V_i. CVP's condition must state that scheduler variable Sc points to module M_i and CVP's valuation is the expression mentioned in $X_i\} \cup Sc_x$.

The main feature of the SMV-intermediate to SMV-flat translation is that the possibly multiple next declarations for each single variable are combined into a single next declaration which depends on the scheduler variable. Before actually proving that the SMV to SMV-flat translation results in a model that is in some way equivalent to the original SMV-input model we first take a look at the induced Transition Systems by SMV models. In order to do so we first describe how the SMV state vector and SMV states are defined as well as what a Cartesian product is.

Definition 42 (SMV state vector and states)
A state vector is a list of slots. Each slot is occupied by a single variable. The state vector can be denoted as $\langle x_1, x_2, x_3, ..., x_n \rangle$ with x_j being the slot for variable j. A state in a Transition System is denoted

$(v_1, v_2, v_3, ..., v_n)$ and consists of the same list of slots occupied by values of the variables present in the state vector.

Definition 43 (Cartesian product)
The product of set X and set Y denoted $X \times Y$ is the set that contains all ordered pairs $\{(x, y) \mid x \in X \wedge y \in Y\}$. The product of three sets X, Y and Z is: $(X \times Y) \times Z$ etcetera.

In an SMV module the state space is discovered as follows: Starting from the initial states as current state, the system first chooses one of the available asynchronous modules (or the synchronous plus "main" module) for execution. It then expands the current state by only applying the **next declarations** found in the module(s) selected for execution. Variables that do not have a next declaration in the selected module(s) do not change in the set of next states [10]. Let us now formally define how an SMV model induces a Transition System.

Definition 44 (Induced Transition System by an SMV model)
Let an SMV model M have N state variables (not define macros) denoted $x_1, x_2, ..., x_N$. Let the sets of possible values per variable be denoted $r_1, r_2, ..., r_N$. Let the sets of possible initial values per variable be denoted $i_1, i_2, ..., i_N$. Let *choices* be a set of choices with one choice for each asynchronous module (plus one choice for the combination of the synchronous modules and the "main" module) in the SMV model. Let the sets of next state values per variable with current state S and module(s) choice C be denoted $n_1(S, C), n_2(S, C), ..., n_N(S, C)$. Note that if the variable does not change for current state S and choice C that the set returned is a singleton set with as its element the value of the variable in current state S.

The induced Transition System $TS = \langle S, I, T \rangle$ by M is defined as follows:

- $S = r_1 \times r_2 \times ... \times r_N$
- $I = i_1 \times i_2 \times ... \times i_N$
- $T = \{(s_1, s_2) \mid \exists C \in \mathit{choices} \; . \; s_2 \in n_1(s_1, C) \times n_2(s_1, C) \times ... \times n_N(s_1, C)\}$

Note that transition relation is left-total as any state that has no changing variables any-more will always have a self-loop.

2.4 SMV to SMV-flat equivalence

In this Section will define what it means for two models, one described in the SMV-input model and one in the SMV-flat model to be equivalent. Our goal is to prove that reachability is preserved in the SMV to SMV-flat translation. The intuition is that reachability is indeed preserved because any sequence of scheduler choices made in the SMV-input model can also be made in the SMV-flat model. The SMV-flat model also does not invent any new sequences of scheduler choices. In order to formalise and prove that reachability is preserved we first take a look at an example of a state space of an SMV-input model and its translated SMV-flat model's state space.

Figure 2.4 shows on the left an improvised state space of an SMV-input model. The SMV-input model has two asynchronous modules named with $P1$ and $P2$. Each transition is labelled with the module that was selected for execution while taking that transition. On the right the translated SMV-flat state space

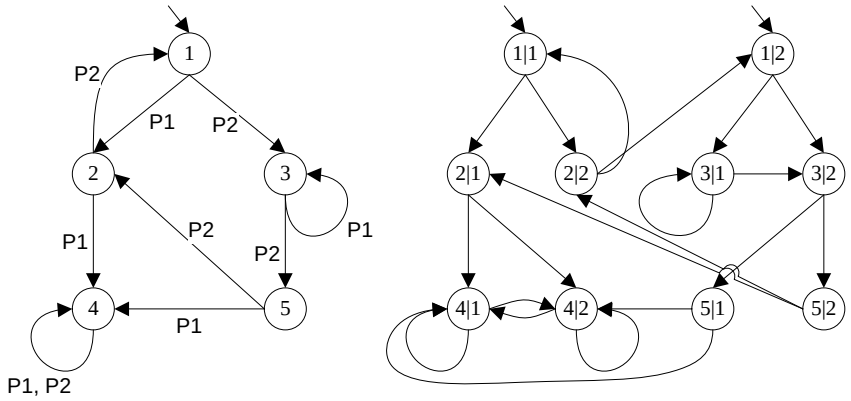


Figure 2.4: An example of an SMV-input model state space (left) and its translated state space.

is shown. The first number in the state indicates the corresponding SMV-input state. The second number shows the value of the scheduler variable which was determined in the previous state. Defining two states to be equivalent if they have the same state number (so not looking at the scheduler variable), we can make a number of observations:

- All SMV-input states exist two times in the SMV-flat state space. In fact, the multiplication factor is equal to the number of scheduler choices.
- Any SMV-input state can simulate the behaviour of any of the equivalent states in the SMV-flat model.
- The combination of the two equivalent states in the SMV-flat model can simulate the behaviour of the equivalent state in the SMV-input model.

It is clear that states in the SMV-input model and states in the SMV-flat model can only be compared to each other if the scheduler variable is not considered. Two states are thus equivalent if they have the same value for all variables in the state vector while not considering the scheduler variable. This notion of state equivalence is defined next:

Definition 45 (State equivalence relation for SMV to SMV-flat)
Let M_i be an SMV-input model and $TS_i = \langle S_i, I_i, T_i \rangle$ the induced Transition System. Let M_f be an SMV-flat model and $TS_f = \langle S_f, I_f, T_f \rangle$ the induced Transition System. Let M_i have N variables and let M_f have N variables plus the scheduler variable denoted Sc, with valuation v_sc. Let $s_i \in S_i$ and $s_f \in S_f$, then states s_i and s_f are equivalent if $(s_i, s_f) \in stateEq_{flat}$. The relation $stateEq_{flat} \subseteq S_i \times S_f$ is defined as follows:

$$stateEq_{flat} = \{((v1_i, v2_i, ..., vN_i), (v1_f, v2_f, ..., vN_f, v_sc)) \mid v1_i = v1_f, v2_i = v2_f, ..., vN_i = vN_f\}$$

SMV-input's ability to simulate SMV-flat and SMV-flat's ability to simulate SMV-input suggests a bisimulation relation. This cannot be a normal bisimulation relation as single states in SMV-flat cannot

simulate their equivalent states in the SMV-input model. Look for example at state 1|1 in Figure 2.4, it needs the help of state 1|2 in order to simulate all behaviour of state 1 in the SMV-input model. To that end the bisimulation must relate states in the SMV-input model to groups of states in the SMV-flat model. Those groups of states are called equivalence classes and are defined next.

Definition 46 (Equivalence class function for states in an SMV-flat model)
Let M_i, TS_i, M_f and TS_f be defined as in Def. 45. For any state $s_f \in S_f$ it's equivalence class function denoted $Eclass(s_f) : S_f \to 2^{S_f}$ is defined as follows:

$$Eclass(s_f) = \{s_e \in S_f \mid \exists s_i \in S_i.(s_i, s_f) \in stateEq_{flat} \wedge (s_i, s_e) \in stateEq_{flat}\}$$

With the definitions of the state equivalence relation and the equivalence classes we can now define the desired bisimulation relation. For our purposes we say that the equivalence-class function is only applied to the second Transition System. The definition would be mathematically cleaner if we would define two equivalence-class functions, one for each Transition System. We could then set the equivalence-class function for the SMV-input model to just return the input state (each state is its own equivalence class). Our definition however does not incorporate two equivalence-class functions as to prevent an even more complex definition.

Definition 47 (Equivalence-class Bisimulation Relation)
Given two Transition Systems $TS = \langle S, I, T\rangle$ and $TS' = \langle S', I', T'\rangle$ with equivalence class function $e' : S' \to 2^{S'}$, a relation $R \subseteq S \times S'$ is an equivalence-class bisimulation relation if $(s, s') \in R$ implies

- if $s \to t \in T$ then $\exists u \in e'(s') \wedge \exists t' \in S' \;.\; \forall z \in e'(t') \;.\; u \to z \in T' \wedge (t, z) \in R$.
- if $s' \to t' \in T'$ then $\exists t \in S.s \to t \in T \wedge (t, t') \in R$

Definition 48 (Equivalence-class bisimilar)
Two Transition Systems $TS = \langle S, I, T\rangle$ and $TS' = \langle S', I', T'\rangle$ are equivalence-class bisimilar if there exists a equivalence-class bisimulation relation R and an equivalence-class function e' such that:

$$\forall s_0 \in I \;\exists s'_0 \in I' \;.\; \forall z \in e'(s'_0) \;.\; (s_0, z) \in R$$
$$\forall s'_0 \in I' \;\exists s_0 \in I \;.\; (s_0, s'_0) \in R$$

2.5 SMV to SMV-flat proof

Using the definitions for SMV-input, SMV-intermediate and SMV-flat (Definitions 34,35,38) and the translations from SMV-input to SMV-intermediate and from SMV-intermediate to SMV-flat (Definitions 37,41) we will now prove that any SMV-flat model that is a result of the SMV-input to SMV-flat translation applied to an SMV-input model is equivalence-class bisimilar to that SMV-input model. The sought after relation R is the state equivalence relation $stateEq_{flat}$ (Def. 45) and the used equivalence-class function is the $Eclass$ function (Def. 46).

Theorem 49
Let M_i be an SMV-input model and let $M_f = \langle Vsf, Dsf, Isf, Xsf\rangle$ be an SMV-flat model that is obtained

from M_i by first applying the SMV-input to SMV-intermediate translation and by then applying the SMV-intermediate to SMV-flat translation. Let $TS_i = \langle S_i, I_i, T_i \rangle$ and $TS_f = \langle S_f, I_f, T_f \rangle$ be the Transition Systems induced by M_i and M_f respectively. Then the following must hold:

TS_i is equivalence-class bisimilar to TS_f under equivalence-class function $Eclass$.

Proof

The sought after equivalence-class bisimulation relation R is the $stateEq_{flat}$ relation. In order to prove that TS_i and TS_f are equivalence-class bisimilar we will first show that states in TS_i and TS_f have a comparable state vector as otherwise we cannot even apply the $stateEq_{flat}$ relation to them. As such, we will show that the state vector of SMV-flat states is equal to that of the SMV-input states with the exception of the introduction of the scheduler variable. Afterwards, we will show that $stateEq_{flat}$ is an equivalence-class bisimulation relation for the TS_i and TS_f transition systems by proving the conditions on the initial states (Def. 48) and the conditions on all states in the relation (Def. 47) to hold.

Let M_i have N_i state variables denoted $xi_1, ..., xi_{Ni}$. Let the state vector for states in TS_i be denoted as $SV_i = \langle xi_1, ..., xi_{Ni} \rangle$. Let N_f, $xf_1, ..., xf_{Nf}$ and $SV_f = \langle xf_1, ..., xf_{Nf} \rangle$ be defined analogously for model M_f.

Proof of: $SV_f = \langle xi_1, ..., xi_{Ni}, Sc \rangle$

The SMV-input model contains a number of **variable definitions**. Those variables are copied without change to set Vs_i by the SMV-input to SMV-intermediate translation. For each variable $Vs = \langle M_i, N_i, T_i \rangle$ the parent module is recorded in M_i, the variable name in N_i and its type in T_i. The SMV-intermediate to SMV-flat translation copies the set Vs_i to set Vs_f while applying the function $rename$ on each element. The translation also adds the scheduler variable Sc to set Vs_f. The $rename$ function embeds the variable's parents module into the variable's name but otherwise leaves the variable unchanged. Therefore, as no variables have been created or lost during the translation other than that Sc has been added, and as the $rename$ function only changes the name of the variables but not its type and or any associated init or next expressions, it must be the case that $SV_f = \langle xi_1, ..., xi_{Ni}, Sc \rangle$.

Proof of: $\forall s_{0i} \in I_i \; . \; \exists s_{0f} \in I_f \; . \; \forall z \in Eclass(s_{0f}) \; . \; (s_{0i}, z) \in stateEq_{flat}$

Let the sets of possible initial values per variable in the input model be denoted $i_{1i}, i_{2i}, ..., i_{Ni}$ and the sets of possible initial values per variable in Vsf be denoted $i_{1f}, i_{2f}, ..., i_{Nf}$.

The SMV-input model contains a number of **init declaration**s. These init declarations are copied to set Is_i by the translation from SMV-input to SMV-intermediate. Each init declaration is represented in the set Is_i by a single element $I_i = \langle M_i, V_i, subst(E_i) \rangle$. As each variable has only a single init declaration associated with it (Def. 28), we conclude that each variable is represented once in set Is_i. The SMV-input to SMV-intermediate translation applies function $subst$ to expression E_i, which does not change the resulting value(s) of the expression as function $subst$ only replaces symbolic names with the expressions that they point to. The translation from SMV-intermediate to SMV-flat copies set Is_i to set Is_f while applying the $rename$ function to each element. This function only changes the name of the represented variable but does not otherwise change the information in any way. The translation also adds an initial expression for the scheduler variable Sc to set Is_f. Therefore,

- $I_i = i_{1i} \times i_{2i} \times ... \times i_{Ni}$
- $I_f = i_{1f} \times i_{2f} \times ... \times i_{Nf} \times Sc_i$

Note that the order of the variables in I_f is not necessarily the same as in I_i, but as we showed that $i_{1i}, i_{2i}, ..., i_{Ni}$ and $i_{1f}, i_{2f}, ..., i_{Nf}$ consists of the same variables under the renaming function, showing that it must be possible to re-arrange the variables such that $i_{1i} = i_{1f}, i_{2i} = i_{2f}, ..., i_{Ni} = i_{Nf}$. For the rest of the proof we assume that the variables in the SMV-input and SMV-flat models are ordered in the same way.

Denoting the number of scheduler choices as *choices*, $I_f = \{(I_i \times C) \mid C \in choices\}$, it becomes clear that for every initial state in the SMV-input model there must exist an initial state in the SMV-flat model for which itself and all of the states in its equivalence-class ($Eclass$) are related to the previously mentioned SMV-input state by $stateEq_{flat}$.

Proof of: $\forall s_{0f} \in I_f \; . \; \exists s_{0i} \in I_i \; . \; (s_{0i}, s_{0f}) \in stateEq_{flat}$
$I_f = \{(I_i \times C) \mid C \in choices\}$ shows that all initial states in I_f have an initial state counterpart in I_i which it is related to by relation $stateEq_{flat}$.

Let $s_i, t_i \in S_i$, $s_f, t_f \in S_f$ and $(s_i, s_f) \in stateEq_{flat}$.

Proof of: if $s_i \rightarrow t_i \in T_i$ then $\exists u \in Eclass(s_f) \wedge \exists t_f \in S_f \; . \; \forall z \in Eclass(t_f) \; . \; u \rightarrow z \in T_f \wedge (t, z) \in stateEq_{flat}$.
Looking at how an SMV model induces a Transition System (Def. 44) we see that the transition $s_i \rightarrow t_i$ was made under the choice of one of the asynchronous modules (also counting the main module and all synchronous modules as a single asynchronous module). Let this particular chosen asynchronous module be denoted as c. Now, seeing as that $(s_i, s_f) \in stateEq_{flat}$, we know that the equivalence class of s_f denoted $Eclass(s_f)$, contains all states that are all related to s_i according to relation $stateEq_{flat}$. By construction of the scheduler variable Sc we know that it's enumeration type contains a value for each asynchronous module (plus 1) in the original SMV-input model. Therefore, no matter what the choice c was, that choice is represented by one of the states in $Eclass(s_f)$. Therefore we have proven the existence of state u that has the same explicit scheduler choice as the transition from $s_i \rightarrow t_i$ made implicitly. Now we still have to show that: $\exists t_f \in S_f \; . \; \forall z \in Eclass(t_f) \; . \; u \rightarrow z \in T_f \wedge (t, z) \in stateEq_{flat}$.

Now let's take a look at the transition relations of both TS_i and TS_f. Both transition relations are defined as follows:
- $T = \{(s_1, s_2) \mid \exists C \in choices \; . \; s_2 \in n_1(s_1, C) \times n_2(s_1, C) \times ... \times n_N(s_1, C)\}$

In more detail, we know that for the transition between $s_i \rightarrow t_i$ we know that we made the choice called c, and we know that there exists a state in the SMV-flat Transition System that has this choice embedded in it's scheduler variable value. Let Sc_c denote the symbolic constant in the scheduler's enumeration type that represents chosen module c:
- $t_i \in n_1(s_i, c) \times n_2(s_i, c) \times ... \times n_N(s_i, c)\}$
- $\exists C \in choices \; . \; t_f \in n_1(s_f, C) \times n_2(s_f, C) \times ... \times n_N(s_f, C) \times Sc_c\}$

As the SMV-flat model only consists of a single synchronous module there can only be one choice C called m (for "main"):
- $t_f \in n_1(s_f, m) \times n_2(s_f, m) \times ... \times n_N(s_f, m) \times Sc_c\}$

Now we show that the sets of next state values for the individual variables are the same in the SMV-input and SMV-flat model: Looking at the **next declaration**s we see that the translation from SMV-input to SMV-intermediate copies the next declarations to the set Xs_i. The function *subst* is applied to the expressions but as discussed before that does not change the expressions. The SMV-intermediate to SMV-flat translation merges the next expressions that are concerning the same variable by grouping them in a single case-esac expression with each their own condition valuation pair. The condition states that the correct asynchronous module is chosen and the valuation is the original next expression. All created case-esac statements (one per variable) are put into set Xs_f and the next expression for the scheduler variable is also added. Therefore, with implicit scheduler choice c in the SMV-input model and explicit scheduler choice Sc_c made in the SMV-flat model, the created case-esac statements will make sure that the correct next declaration's expression is selected for next state value computation. As such we have proven that each set of next state values for the individual variables are actually equal, and so there must exist a state $t_f \in n_1(s_f, m) \times n_2(s_f, m) \times ... \times n_N(s_f, m) \times Sc_c\}$ for which it holds that $(t_i, t_f) \in stateEq_{flat}$.

Lastly we need to prove that there exists a translation form state u not only to t_f, but to every state in $Eclass(t_f)$. This is however trivial as the only difference between the states in $Eclass(t_f)$ is the scheduler variable whose next declaration states that it's next state valuation is chosen non-deterministically ensuring that all states in $Eclass(t_f)$ are generated. We have also already shown that for each state in $Eclass(t_f)$ called z it holds that $(t, z) \in stateEq_{flat}$, because except for the scheduler variable their individual variable values are exactly the same.

Proof of: if $s_f \to t_f \in T_f$ then $\exists t_i \in S_i.s_i \to t_i \in T_i \wedge (t_i, t_f) \in stateEq_{flat}$
Having already shown that the SMV-input model has exactly the same implicit scheduler choices available as the SMV-flat model does in an explicit way, and that the expressions for next state value computation are then equal for each individual variable, we know that every transition in the SMV-flat model can also be simulated in the SMV-input model.

Note that the definition of the Equivalence-class Bisimulation Relation (Def. 47) and the conditions on the initial states in definition equivalence-class bisimilar (Def. 48) ensure that all states in reachable equivalence classes are in fact also reachable themselves, making sure that the reachability property is preserved.

2.6 The mCRL2 language

In this section we will formalise the mCRL2 language. We will only describe the subset of the language that shows up in the results of our SMV-flat to mCRL2 translations (a small subset).

Definition 50 (mCRL2 model)
An mCRL2 model is a collection of the following items:

1. A number of **sort specification**s.
2. A number of **mapping specification**s.
3. A single **action specification**.
4. A single **process specification**.
5. A single **init statement**.

Definition 51 (mCRL2 sort specification)
The mCRL2 sort specification is a definition of an enumeration type. The sort specification contains a name for the enumeration type and it contains a list of **mCRL2 symbolic constants** that together form the enumeration type.

Definition 52 (mCRL2 symbolic constant)
An mCRL2 symbolic constant is a string that abides by the rules of the **mCRL2 variable identifier** which can be used as an enumeration symbol.

Definition 53 (mCRL2 integer constant)
An mCRL2 integer constant is an integer that can be used as a terminal symbol in **expression**s.

Definition 54 (mCRL2 mapping specification)
The mCRL2 mapping specification is mCRL2's variant of a function. A mapping expression consists of three parts called map, var and eqn. The "map" part consists of the name of the macro and its type. Both the function's parameter types and return type is defined. The "var" part consists of a list of function parameters, defined as their internal function name and their type. The "eqn" part lists a number of equations that define the behaviour of the function.

Definition 55 (mCRL2 action specification)
The mCRL2 action specification declares which **action**s are used in this mCRL2 model.

Definition 56 (mCRL2 process specification)
An mCRL2 process specification is a 3-tuple (denoted $\langle N_a, Vs_a, Sums_a \rangle$) consisting of the following items:

1. A process name contained in a **variable identifier**.
2. A list of **data variable**s.
3. A number of **summand**s.

Definition 57 (mCRL2 init specification)
The mCRL2 init specification declares which **action**s are allowed to be executed, which process is selected to be executed and the initial values for the variables of the mentioned processes. It is not allowed to

specify multiple initial values for a single variable and as such it is not possible to create multiple initial states.

In the **init specification** it is possible to refer to the start process as an parallel composition of multiple processes. As we only have a single process in our mCRL2 models we will not look into that part of mCRL2 any further.

Definition 58 (mCRL2 data variable)
An mCRL2 data variable is a state variable and is denoted by its **mCRL2 variable identifier** and its **mCRL2 variable type**.

Definition 59 (mCRL2 variable identifier)
An mCRL2 variable identifier is an string that does not begin with a number, but that otherwise may consist of A-Z, a-z, 0-9 and the special symbols “”’ and “_”.

Definition 60 (mCRL2 variable type)
An mCRL2 type is a description of a variables type, being either “Bool”, “Int”, or “User defined sort”. The “Bool” type is the standard boolean type, “Int” the integer type and the “User defined sort” is a type that can be defined by enumerating its possible valuations. The “User defined sort” type is therefore effectively an enumeration type and will denoted “Enumeration” from now on.

Definition 61 (mCRL2 summand)
An mCRL2 summand consists of an optional **condition expression**, an **action** and a **process-call expression** and is denoted $\langle cond, action, pcall \rangle$.

Definition 62 (mCRL2 condition expression)
An mCRL2 condition expression consists of a **mCRL2 expression** which must have the Bool type. The calculation of the condition expression related to a state s is denoted $cond(s)$ and has a boolean result.

Definition 63 (mCRL2 action)
mCRL2 actions are used to label transitions with the name of the action. The mCRL2 language does not allow for the action to be absent. As we are not interested in using actions we use the improvised dummy action: “noAction”.

Definition 64 (mCRL2 process-call expression)
An mCRL2 process-call expression redirects model execution to a process mentioned by a **variable identifier** and defines next state valuations for that process’s variables by listing expressions that determine their next state valuations. Any variables for which the process-call does not mention the next state valuation are left unchanged. The application of a process-call expression to a current state s is denoted $pcall(s)$ and results in a next state of state s.

Definition 65 (mCRL2 expression)
An mCRL2 expression can be seen as a tree, with several nodes that have a certain number of expressions as its children (denoted #). An expression may begin with any of the available nodes. The available operator

nodes and terminal nodes are described by the table below. The "any" type stands for the three types: Bool, Int and Enumeration.

Table 2.2: mCRL2 expression operators

Operator	#	Result Type	Child type	Description
(..)	1	any	any	precedence
! ..	1	Bool	Bool	logical negation
- ..	1	Int	Int	unary minus
.. &&, || ..	2	Bool	Bool	logical and, logical or
.. ==, != ..	2	Bool	any	equality, inequality
.. <, >, <=, >= ..	2	Bool	any	less, greater, .. or equal, .. or equal
.. +,- ..	2	Int	Int	addition, subtraction
.. *,/,mod ..	2	Int	Int	multiplication, division, remainder
.. in ..	2	Bool	Int, enumeration	list inclusion
if(.. , .. , ..)	3	any	**	if-then-else
[..]	*	List of any	any, but all the same	list creation

** The type of the children of the if-then-else expressions differs per child: The first child must have a boolean type, and the other two may have any type.

Terminals are either **mCRL2 symbolic constant**s, **mCRL2 integer constant**s, or **mCRL2 variable identifier**s that point to variables declared in a **process declaration** and or available **mapping expression**s. These terminals have respectively the types enumeration, integer, the type of the pointed-to variable and the result type of the pointed-to mapping expression.

2.7 SMV-flat to mCRL2 translation

When translating from SMV-flat to mCRL2 there are multiple things to consider that do not have to do with any specific translation from SMV-flat to mCRL2. This Section covers how types and expressions are translated, how SMV define macros can be translated and how we can support multiple initial states in mCRL2 models.

Definition 66 (Translation between SMV and mCRL2 variable types)
SMV variable types (Def. 24) are translated to mCRL2 variable types (Def. 60) following the table shown below:

Table 2.3: Translation from SMV type to mCRL2 type

SMV type	mCRL2 type
Boolean	Bool
Integer range	Int
Enumeration	User defined sort
module type	Not applicable

The translation from the **SMV integer range** to mCRL2's Int type is potentially problematic: An integer range is a clearly defined part of the integer range whereas mCRL2's Int type represents the whole integer range. As we are only interested in SMV models that are correct in the sense that the calculated variable valuations stay within their defined integer range, we might just as well translate towards a broader type without encountering any problems. The SMV module type is not applicable because any SMV-flat model only consists of a single module and therefore cannot have variables with an SMV module type.

2.7.1 Translation of SMV expressions to mCRL2 expressions

Definition 67 (Translation between SMV and mCRL2 expressions)
The subset of supported **SMV expressions** (Def. 33) has been selected such that most operators in that subset have an mCRL2 counterpart with the same semantics. The **case-esac expression** does not have an mCRL2 counterpart but can be transformed into other operators which are translatable. The needed transformation is described in Section 2.7.1. The **next valuation expression** does not have to be translated as it is already substituted away by the SMV-input to SMV-intermediate translation (Def. 37). For the other SMV operators the proposed translation is described in table 2.4

Table 2.4: Translation from SMV to mCRL2 expression operators

SMV Operator	mCRL2 Operator	Description
(..)	(..)	precedence
! ..	! ..	logical NOT
- ..	- ..	unary minus
.. &, | ..	.. &&, || ..	logical AND, logical OR
.. =, != ..	.. ==, != ..	equality, inequality
.. <, >, <=, >= ..	.. <, >, <=, >= ..	less, greater, .. or equal, .. or equal
.. +,- ..	.. +,- ..	addition, subtraction
.. *,/,mod ..	.. *,/,mod ..	multiplication, division, remainder
.. in ..	.. in ..	set inclusion
.. ? .. : ..	if(.. , .. , ..)	if-then-else
{ .. }	**	non-deterministic choice
{ .. }	[..]	In a set context: creation of a set

** SMV supports non-deterministic valuations of variables when defining their next state valuation in a **next declaration**. mCRL2 handles non-determinism in a different way by influencing the execution of the mCRL2 model by specifying multiple **mCRL2 summand**s from which the system may non-deterministically choose. The translation of non-deterministic choice expressions to mCRL2 summands is described in the translation from SMV-flat to mCRL2 All-in-1 (Def. 73) and in the translation from SMV-flat to mCRL2 1-by-1 (Def. 80).

The supported expression terminals in SMV expressions are **SMV symbolic constant**s, **SMV integer constant**s and **SMV variable identifier**s. The SMV variable identifiers can either point to **SMV variable declaration**s and or **SMV define declaration**s. Translating the SMV symbolic constants and SMV variable identifiers that point to SMV variable declarations happens by using Definition 68 which only possibly changes the name of the symbolic constant. SMV integer constants can be translated without

any change. The translation of SMV variable identifiers that point to a SMV define declaration depends on the specific translation chosen. Two translations have been devised which are described in Section 2.7.2.

Note that we translate SMV set creations to mCRL2 list creations. As SMV only supports the use of such sets as the right hand side of the **in expression** it becomes clear that it does not matter if we look at the set of valuations as a set or a list; The "in expression" only determines if a certain element is in a group of element. Such an operation does not care about the order of the element in the group.

In order to support the use of **mCRL2 mapping specification**s, which are effectively function declarations it is also possible for the mCRL2 expression to be a terminal that points to this mapping expression by stating its name and arguments for all its parameters.

case-esac to if-then-else

SMV case-esac expressions do not have a counterpart **mCRL2 expression** construct. mCRL2 does however support the use of the if-then-else construct. When looking at the definition of the case-esac expression (Def. 33) we see that the evaluation of a case-esac expression resembles that of a nested if-then-else structure. This becomes more clear when we look at a small example:

```
1 case
2   condition1 : valuation1 ;
3   condition2 : valuation2 ;
4   ...
5 esac;
```

The above shows a case-esac expression which is to be evaluated as follows: First *condition*1 is evaluated. If *condition*1 evaluates to TRUE then the case-esac expression evaluates to *valuation*1. If not, we look at the second pair: If *condition*2 evaluates to TRUE then the case-esac expression evaluates to *valuation*2 (and so forth). As such, it becomes trivial that this case-esac expression can be translated to the following if-then-else expression:

```
1 if condition1 then valuation1 else (if condition2 then valuation2 else (if ...)).
```

For any finite amount of condition valuation pairs in a case-esac expression there comes a moment that this translation has processed the last condition valuation pair after which one "else" expression remains empty (as there is no new condition valuation pair's if-then-else representation to put there). This is however not a problem as it is allowed to fill in anything: SMV dictates that for all possible current states one of the conditions of the case-esac expression must be true and as such this last else expression will never be reached.

union expression

The **SMV union expression** is used in the SMV modelling language to determine a set of values, which once created can be used in combination with the **SMV in expression**. The resulting set can also be used when assigning a next state value to a variable signifying a non-deterministic choice from all the values in the set. The translation of union expressions therefore depends on the situation: If the result of the union is used as a non-deterministic choice then the union expression is translated to an SMV non-deterministic choice expression. If the result of the union is used as a set in combination with an "in expression", then it is marked as such and will be translated to an mCRL2 list.

Translation between variable identifiers

Definition 68 (Translation between SMV and mCRL2 variable identifiers)
SMV variable identifiers (Def 25) are translated to mCRL2 variable identifiers (Def 59) by renaming the special symbols that are allowed in SMV but not in mCRL2. The symbols in question are "$","#", and "-". These symbols can be replaced in any string used as an identifier by replacing them with for example their names "dollar", "hashtag" and "minus".

2.7.2 Translation of SMV define macros

Definition 69 (Translation of SMV define macros by substitution)
Translation of SMV define macros by substitution works by substituting the expression that expresses how the define macro is calculated for every occurrence of a SMV variable identifier that points to the define macro in every expression found in the SMV model. The calculating expression can be found in the **SMV define declaration**.

Definition 70 (Translation between SMV define macros and mCRL2 mapping specification)
Translation between **SMV define macro**s and **mCRL2 mapping specification**s works by generating a mapping specification for each SMV define declaration and then changing all expressions that point to the define declaration in order to correctly call the mCRL2 mapping function.

Let the mapping expression for variable $D_f = \langle N_f, E_f \rangle \in Ds_f$ be defined as $mapping(D_f) = \langle map, var, eqn \rangle$ with:

- $map = \langle N_f, \{T_f\} \rangle$. The set of types $\{T_f\}$ consists of the types of the variables mentioned in var plus the return type of D_f. The return type of D_f is equal to the type of the top expression node of expression E_f (see Def. 33).
- $var = \{\langle N_d, T_d \rangle \mid E_f$ depends on variable $V_f = \langle N_d, T_d \rangle\} \in Vs_f$.
- $eqn = E_f$ with any references to other define macros augmented with their correct mCRL2 function call (the depending variables must be given as function arguments).

Define macro D_f depends on a state variable $V_f \in Vs_f$ if the value of that variable in the current state is needed for the computation of expression E_f. This is the case if E_f contains a **symbolic constant** that points to variable V_f, or if V_f is depended upon by any define macro mentioned in E_f.

Let's look at an example of the define macro to mapping expression translation. Below an example is shown of three define macros defined within an SMV model of an improvised game. The model has state variables *state* and *score* (not depicted) and define macros *start*, $finish$, *middle*. The *start* macro states that the game is started when both the *state* and *score* variables equal 0. The $finish$ macro states that the game is finished if either the *state* is 6 or if the *score* is bigger than 10. The *middle* define macro defines when the game is neither just started nor finished. In order to do so it refers to the definitions of *start* and $finish$.

```
1  DEFINE
2    start := state = 0 & score = 0;
3    finish := state = 6 | score > 10;
4    middle := !start & ! finish;
```

The translation of these define macros is depicted below. The translation for the *start* and $finish$ define macros is fairly straight foreward: They both depend on *state* and *score* which are of integer type and the function results in a boolean type because the top expression is "and" resp. "or". The translation of *middle* is more involved as it depends on *start* and $finish$. First it is determined that *start* and $finish$ depend on *state* and *score* and as such so does *middle*. As such *state* and *score* are added to the mapping expression. Lastly the expression of the *middle* define macro is adapted to correctly call upon the *start* and $finish$ mapping expressions. This same adaption must be done for all expressions in the SMV-flat model. In those expressions the needed variable are however always available by their symbolic name as we only use a single mCRL2 process.

```
1  map start: Int # Int -> Bool;
2  var state : Int;
3      score : Int;
4  eqn start(state,score) = state = 0 & score = 0;
5
6  map finish: Int # Int -> Bool;
7  var state : Int;
8      score : Int;
9  eqn finish(state,score) = state = 6 | score > 10;
10
11 map middle: Int # Int -> Bool;
12 var state : Int;
13     score : Int;
14 eqn middle(state,score) = !start(state,score) & ! finish(state,score);
```

Except for the creation of a mapping specification for each SMV define macro we must also adapt any found variable identifiers in expressions that point to SMV define macro's to incorporate the correct function call. All depending state variables must be added to the function call which should be no problem as we have only a single mCRL2 process and therefore all variables are readily available.

2.7.3 Support for multiple initial states

SMV models may have multiple initial states. mCRL2 models however are restricted to a single initial state. The solution deployed to counteract this problem is that all translations from SMV to mCRL2 first generate the set of initial states equal to the SMV set of initial states before continuing to generate the rest of the state space. In order to generate the initial states from the mCRL2 initial state we introduce the special **Initial flag** variable (Def. 71) into the mCRL2 models. The idea behind the initial flag is that the flag can be used to determine a special mode of next state generation in the mCRL2 models: If the initial flag is set to "True" then the next states are all initial states of the SMV-flat model. If the initial flag is set to "False" then the next states are all next states as defined by the SMV-flat model.

Definition 71 (Initial flag)
The initial flag, denoted *If*, is a state variable with the boolean type that is used in mCRL2 models to determine whether a state is the unique initial state or not. To that end, the initial flag is only set to "True" for the initial state in the mCRL2 model. The **init declaration** and the **next declaration** for the initial flag is just the valuation "False", ensuring that any other state then the initial state will be marked

with the information that it is not an initial state. For the initial state the initial flag's valuation is set to "True" in the **mCRL2 init statement**.

2.8 SMV-flat to mCRL2 All-in-1 formalisation

Having investigated multiple parts of the translation from SMV-flat to mCRL2 we are now ready to formalise the All-in-1 translation. The concept behind the All-in-1 translation is that it translates one SMV state to a single equivalent mCRL2 state. The translation is based on the idea that the single mCRL2 process is created with process-call expressions that change all variables at the same time. In order to support non-determinism, all combinations of the non-deterministic choices are computed and that same amount of mCRL2 summands are created each representing a single non-deterministic choice combination. All summands are created without a condition and the dummy action "noAction". The mCRL2 semantics (Def. 74) then dictate that the translated model has to choose between all created summands in a non-deterministic manner. The All-in-1 translation makes use of the *determiniser* function defined next.

Definition 72 (Substitute Nondeterministic and Determiniser function)
The *determiniser* function creates a set of deterministic expressions from an **expression** that may contain **non-deterministic expression**s. It's application is to determinise next state valuation expressions. Let E_n be an expression that possibly contains non-deterministic expressions. E_n may have multiple non-deterministic expressions due to the **case-esac expression**: The case-esac expression may, upon different values for referenced variables in it's conditions, evaluate to different non-deterministic expressions. Let E_n have N different non-deterministic expressions denoted $Ne_1, Ne_2, ..., Ne_N$. Each non-deterministic expression consists of a set of values from which the non-deterministic choice is made, denoted Nsc_1 for Ne_1, Nsc_2 for Ne_2 etcetera. Define function $substN : E \times \{E\} \times \{E\} \to E$ to be a function which substitutes all non-deterministic expressions given in expression set Nse for a set of expressions given in expression set Nsc:

$substN(E_n, Nse, Nsc) = E_n$ with all non-deterministic expressions given in set Nse substituted for their respective expression given in Nsc.

Let the $determiniser : E \to \{E\}$ function be defined as follows:

$$determiniser(E_n) = \{substN(E_n, Nse, Nsc) \mid Nse = \{Ne_1, Ne_2, ..., Ne_N\} \wedge \exists Nsc \in \{Nsc_1 \times Nsc_2 \times ... \times Nsc_N\}\}$$

Let's look at an example of the *determiniser* function before moving on to using it in the All-in-1 translation. The example displayed below is an SMV case-esac expression of which the valuation depends on two variables called *data*1 and *data*2. The expression contains two non-deterministic expressions which are used depending on the values of *data*1 and *data*2.

```
1 case
2   data1 = 1 : {0,1};
3   data2 = 1 : {2,3};
4   TRUE : 0;
5 esac;
```

Below we see the resulting expressions from the *determiniser* function. The function generates four expressions because there are two non-deterministic choices with both two options. The results of the *determiniser* function represents all possible combinations of choices for the non-deterministic expressions.

```
case
  data1 = 1 : {0};
  data2 = 1 : {2};
  TRUE : 0;
esac;
```

```
case
  data1 = 1 : {1};
  data2 = 1 : {2};
  TRUE : 0;
esac;
```

```
case
  data1 = 1 : {0};
  data2 = 1 : {3};
  TRUE : 0;
esac;
```

```
case
  data1 = 1 : {1};
  data2 = 1 : {3};
  TRUE : 0;
esac;
```

Definition 73 (Translation between SMV-flat to mCRL2 "All-in-1")
Using the translation for **variable identifier**s (Def. 68), the translation for **variable type**s (Def. 66), the translation for **expression**s (Def. 67), together with one of the two options for translating **SMV define macro**s (Def. 69, Def. 70), we almost have a complete translation. Only the contents of the **mCRL2 process specification**, the **mCRL2 init specification** and the **mCRL2 action specification** have yet to be determined.

Let the SMV-flat model be denoted $M_f = \langle Vsf, Dsf, Isf, Xsf \rangle$ (Def. 38). Let the mCRL2 process specification be denoted $P = \langle N_a, Vs_a, Sums_a \rangle$, with N_a the name of the process, Vs_a the set of variables and $Sums_a$ a set of summands. Let a summand be denoted as a 3-tuple $\langle cond, action, pcall \rangle$ with *cond* the condition expression, *action* the action and *pcall* the process-call expression. Let a process-call expression be a set of expressions that each represent the next state valuation for a single variable. Let the number of variables in Vsf be denoted N. Let Ei_x and En_x be resp. the init and next state valuation expression for variable V_x described in resp. Isf and Xsf. Let the *determiniser* function (Def. 72) be short-handed to *det*. Let If be the Initial Flag variable, Ei_{if} and En_{if} be resp. the init and next expressions for the initial flag variable. The "All-in-1" translation then translates model M_f to process specification P as follows:

- $N_a =$ "P".
- $Vs_a = Vs_f \cup If$.
- $Sums_a = \{\langle If == True, \text{"}noAction\text{"}, pcall \rangle \mid \exists pcall \in det(Ei_1) \times det(Ei_2) \times ... \times det(Ei_N) \times det(Ei_{If})\}$
  $\cup \{\langle If == False, \text{"}noAction\text{"}, pcall \rangle \mid \exists pcall \in det(En_1) \times det(En_2) \times ... \times det(En_N) \times det(En_{if})\}$

The **mCRL2 action specification** must be created to declare the "noAction" action. The **mCRL2 init statement** must be created with a random (but in range) value for each variable except for the initial flag which must be set to "True". The mCRL2 init statement should also state that the "noAction" action is allowed to be executed.

mCRL2 models originally induce a Labelled Transition System and not a Transition System. The difference is in the presence of labels on the transitions. For our purposes we will define how an mCRL2 model induces a Transition System without any labels which can be achieved by just leaving the mCRL2 labels out of consideration. The semantics presented here are only those of a small subset of the mCRL2 language and are not equal (but for our purposes they are) to the official semantics as presented by the mCRL2 research group.

Definition 74 (Induced Transition System by an mCRL2 model)
Let an mCRL2 model M have a single process with N variables denoted $x_1, x_2, ..., x_N$. Let the range of the variables be denoted $r_1, r_2, ..., r_N$. Let the initial values as defined in the **mCRL2 init statement** per variable be denoted $i_1, i_2, ..., i_N$. Let the mCRL2 process specification of model M be denoted $P = \langle N_a, Vs_a, Sums_a \rangle$ (Def. 73). The induced Transition System $TS = \langle S, I, T \rangle$ by M is defined as follows:

- $S = r_1 \times r_2 \times ... \times r_N$
- $I = \{(i_1, i_2, ..., i_N)\}$
- $T = \{(s_1, s_2) \mid \exists sum = \langle cond, action, pcall \rangle \in Sums_a \ . \ cond(s_1) == \text{“}True'' \wedge s_2 = pcall(s_1)\}$

2.9 SMV-flat to mCRL2 All-in-1 equivalence

The All-in-1 translation has as its main characteristic that all generated **process-call specification**s change all variables at the same time. In contrast to the 1-by-1 translation the All-in-1 translation generates an equivalent mCRL2 state for each SMV-flat state. The equivalence between a state in the SMV-flat model and a state in the All-in-1 model is determined by a state equivalence function, which in this case relates states from the SMV-flat model to their equivalent state in the All-in-1 by adding the Initial Flag variable to the SMV-flat state.

Definition 75 (State equivalence function for SMV-flat to All-in-1)
Let M_f be an SMV-flat model and $TS_f = \langle S_f, I_f, T_f \rangle$ the induced Transition System. Let M_f have N_f state variables. Let M_a be an All-in-1 model and $TS_a = \langle S_a, I_a, T_a \rangle$ the induced Transition System. Let *If* denote the Initial Flag variable. The state equivalence function denoted $stateEq_{All-in-1}$: $S_f \rightarrow S_a$ is defined as follows:

$$stateEq_{All-in-1}((v1_f, v2_f, ..., vN_f)) = (v1_f, v2_f, ..., vN_f, False)$$

The state spaces of the SMV-flat and translated All-in-1 model are equal except for the unique mCRL2 initial state that generates the set of SMV-flat initial states as described in Section 2.7.3. As that is the only difference the equivalence proof for the state spaces we will be split into two parts:

- Prove that the unique mCRL2 initial state correctly generates a set of mCRL2 states that are equivalent to the set of SMV-flat initial states according to $stateEq_{All-in-1}$.
- Pretending that the unique mCRL2 initial state does not exist, and taking the generated set of mCRL2 states that are equal to the set of SMV initial states, prove that the state spaces of the SMV-flat and mCRL2 models are equivalent.

The equivalence used between SMV-flat and All-in-1 models is one that should express that they have an essentially equivalent state space. Such an equivalence is called an Isomorphism and is defined next.

Definition 76 (Isomorphism)
Two Transition Systems TS and TS' are isomorphic if there exists a bijective function $f : S \to S'$ such that

- $\forall s_0 \in I \; \exists s'_0 \in I' \wedge f(s_0) = s'_0$
- $\forall s'_0 \in I' \; \exists s_0 \in I \wedge f(s_0) = s'_0$
- $\forall s_1, s_2 \in S : s_1 \to s_2 \in T \Leftrightarrow f(s_1) \to f(s_2) \in T'$

The way in which the unique mCRL2 initial state should generate a set of mCRL2 states that is equal to the set of initial states from the SMV-flat model is formally expressed in the following definition:

Definition 77 (Correct initial states generation)
Let the unique All-in-1 initial state be denoted s_{0a}. An mCRL2 All-in-1 model with induced Transition System TS_a correctly generates a set of initial states equal to the set of initial states of an SMV-flat model with induced Transition System TS_f if:

- $\forall s_{0f} \in I_f \; \exists s \in S_a \; . \; s_{0a} \to s \in T_f \wedge f(s_{0f}) = s$
- $\forall s \in S_a \mid s_{0a} \to s \in T_a \; \exists s_{0f} \in I_f \wedge f(s_{0f}) = s$

2.10 SMV-flat to mCRL2 "All-in-1" proof

This section presents the proof that the SMV-flat to mCRL2 All-in-1 translation results in an All-in-1 model that is isomorphic to the original SMV-flat model under the assumption that an mCRL2 model is allowed to have multiple initial states which are equal to the SMV-flat initial states. In order to prove that assumption correct, we will first prove that the conditions for correct initial states generation are met. Afterwards we will prove that together with the assumption that the initial states are correctly generated, the two models are isomorphic under the bijective function $stateEq_{All-in-1}$.

Theorem 78
Let M_f be an SMV-flat model and let M_a be a All-in-1 model that is obtained from M_f by applying the SMV-flat to mCRL2 All-in-1 translation. Let $TS_f = \langle S_f, I_f, T_f \rangle$ and $TS_a = \langle S_a, I_a, T_a \rangle$ be resp. the Transition System induced by M_f and M_a. Then the following must hold:

TS_f is isomorphic to TS_a. The isomorphism is given by state equivalence function $stateEq_{All-in-1}$. The isomorhpism holds under the assumption that mCRL2 models are allowed to have multiple initial states, which have to be proved equal to the initial states of the SMV-flat model.

proof

In order to be able to prove anything we first have to confirm that we can use function $stateEq_{All-in-1}$. This function assumes that if the state vector in an SMV-flat model has N_f variables then the translated All-in-1 model has $N_f + 1$ variables of which the last one is of boolean type. The first N_f variables must also all be of equal type.

Let $M_f = \langle Vsf, Dsf, Isf, Xsf \rangle$. Let Vsf consists of N_f state variables denoted $xf_1, ..., xf_{N_f}$. Let the state vector for states in TS_f be denoted as $SV_f = \langle xf_1, ..., xf_{N_f} \rangle$. Let Ei_x and En_x be resp. the init and next state valuation expression for variable V_x described in resp. Isf and Xsf. Let the mCRL2 process specification of the M_a model be denoted $P = \langle N_a, Vs_a, Sums_a \rangle$. Let the state vector for model M_a be denoted SV_a. Let the Initial Flag variable be denoted *If*.

Proof of: $SV_a = \langle xf_1, ..., xf_{Nf}, If \rangle$. State vectors consist of all state variables in a model. For M_f those are the variables in Vsf and for M_a those are the variable in Vs_a. The SMV-flat to mCRL2 All-in-1 translation states that $Vs_a = Vs_f \cup If$. Therefore, it must be the case that SV_a consists of the same variables as SV_f, with the same types, and with If added. Therefore, $SV_a = \langle xf_1, ..., xf_{Nf}, If \rangle$.

Let's now take a look at the conditions for correct initial states generation as defined in Def. 77.

Proof of: $\forall s_{0f} \in I_f \ \exists s \in S_a \ . \ (s_{0f}, s) \in T_f \wedge stateEq_{All-in-1}(s_{0f}) = s$ and $\forall s \in S_a \mid (s_{0a}, s) \in T_a \ \exists s_{0f} \in I_f \wedge stateEq_{All-in-1}(s_{0f}) = s$.
The SMV-flat to mCRL2 All-in-1 translation states that the mCRL2 init statement is created with random values for each variable except for variable If which is set to "*True*", which is therefore exactly the contents of state s_{0a}. The definition of the Induced TS by an mCRL2 model states that:

$$T_a = \{(s_1, s_2) \mid \exists sum = \langle cond, action, pcall \rangle \in Sums_a \ . \ cond(s_1) == \text{“}True\text{”} \wedge s_2 = pcall(s_1)\}$$

As such, the next states of state s_{0a} are those states created by summands whose condition under current state s_{0a}, denoted $cond(s_{0a})$, evaluates to "True". Looking at the translation, the only summands that are to be considered for next state generation are those that have a condition that expresses that the Initial Flag is equal to "True":

$$\{\langle If == True, \text{“}noAction\text{”}, pcall \rangle \mid \exists pcall \in det(Ei_1) \times det(Ei_2) \times ... \times det(Ei_N) \times det(Ei_{If})\}$$

The next state of a single summand is created by applying its process-call expression (denoted *pcall*) to the current state (denoted $pcall(s_{0a})$). The effect of $pcall(s_{0a})$ is that all variable change expressions in *pcall* are applied to s_{0a} in order to generate a next state of s_{0a}.

Looking at the induced Transition System by an SMV model we see that the definition denotes the sets of possible initial values per variable to be $i_1, i_2, ..., i_{Nf}$. It also states that $I_f = i_1 \times i_2, \times ... \times i_n$. These sets of possible initial values per variable must have been generated using the expressions in their **init declarations** which we call Ei_1 for xf_1, Ei_2 for xf_2 etcetera. As the *determiniser*'s function (Def. 72), here short-handed to *det*, is exactly to creates an equivalent set of deterministic expressions from a single possibly non-deterministic expression, we conclude that $i_1 == det(Ei_1)$, $i_2 == det(Ei_2)$ etcetera.

As such, not counting the valuation for the Initial Flag variable in All-in-1 states, we have proven that $I_f == \{s_a \mid (s_{0a}, s_a) \in T_a\}$. As the difference in the presence of the Initial Flag variable is precisely what the $stateEq_{All-in-1}$ function takes care of we have proved that both of the following conditions hold:

$$\forall s_{0f} \in I_f \ \exists s \in S_a . (s_{0f}, s) \in T_f \wedge stateEq_{All-in-1}(s_{0f}) = s$$
$$\forall s \in S_a \mid (s_{0a}, s) \in T_a \ \exists s_{0f} \in I_f \wedge stateEq_{All-in-1}(s_{0f}) = s$$

As the definition of the correct initial state generation implies all conditions on the initial states of the Isomorphism definition we will skip those and move on to the third condition.

Proof of: $\forall s_{f1}, s_{f2} \in S_f : s_{f1} \rightarrow s_{f2} \in T_f \Leftrightarrow stateEq_{All-in-1}(s_{f1}) \rightarrow stateEq_{All-in-1}(s_{f2}) \in T_a$ The transition relation of the SMV-flat model is defined as follows:

$$T_f = \{(s_1, s_2) \mid \exists C \in choices \;.\; s_2 \in n_1(s_1, C) \times n_2(s_1, C) \times ... \times n_N(s_1, C)\}$$

With $n_1(S, C)$ being the result of the next expression of variable xf_1 under current state S and scheduling choice C, $n_2(S, C)$ the same for xf_2 etcetera. *choices* is a set of all choices of asynchronous modules plus one choice for the combination of the "main" module and the remaining synchronous modules. As the SMV-flat module only contains the "main" module only a single choice is possible which that "main" module. As such, the choice is of no influence and can be removed from the definition:

$$T_f = \{(s_1, s_2) \mid s_2 \in n_1(s_1) \times n_2(s_1) \times ... \times n_N(s_1)\}$$

As seen, the transition relation of an All-in-1 model is defined as follows:

$$T_a = \{(s_1, s_2) \mid \exists sum = \langle cond, action, pcall\rangle \in Sums_a \;.\; cond(s_1) == \text{“}True'' \wedge s_2 = pcall(s_1)\}$$

This time however we only deal with summands from the All-in-1 model that have the condition that the Initial Flag is equal to "False". This is because only the unique mCRL2 initial state has an Initial Flag with the value "True", and for this isomorphism we assume that generation of the correct set of initial states has already been proven (which we have). The created summands in the SMV-flat to mCRL2 All-in-1 translation that have the correct condition are:

$$\{\langle If == False, \text{“}noAction'', pcall\rangle \mid \exists pcall \in det(En_1) \times det(En_2) \times ... \times det(En_N) \times det(En_{if})\}$$

As with the initial states, we see that $det(En_1) == n_1(s_1)$, $det(En_2) == n_2(s_1)$ and so forth. As such, if we compare states using the $stateEq_{All-in-1}$ state equivalence function, it must be the case that any SMV-flat transition is also created (from the same state to the same state) as an All-in-1 transition and the other way around.

2.11 SMV-flat to mCRL2 "1-by-1" formalisation

The "All-in-1" model determines a next state by looking at the next declarations of all variables at the same time. The "1-by-1" model determines next states by looking at each variable in a separate manner: When determining a next state, each variable gets its own state in which its next valuation is determined. The combination of the transitions between multiple states in the 1-by-1 model (when all variables have been looked at) therefore simulates a single transition in the SMV-flat model. As each variable is calculated separately and as the next state valuation of a variable may depend on other variables we need to keep track of both the newly calculated next state value and the current state value of any variable. We therefore declare the 1-by-1 model to have two variables for each variable in the SMV-flat model. One that records the current state value and one that records the possibly already calculated next state value. After calculating the next value of all variables the 1-by-1 model copies all next state variables to the current state variables before starting repeating the process. Enforcing the 1-by-1 model to determine the next valuations of all variables before starting over is done by a special **counter variable** which is defined next.

Definition 79 (Counter variable)
Let an SMV-flat model have N state variables. The counter variable is used to keep track of the state variable that is to be analysed in the current state. The counter variable therefore has N different values, plus an extra one that signals the special copy transition. Let the counter variable be denoted Cv. Cv is of integer type and ranges from 0 to N. Let the init expression of Cv be the value 0 and let the next expression of the counter variable be $(Cv + 1)$ *mod* N denoted as Cn.

Next we will use the counter variable to formally define the 1-by-1 translation.

Definition 80 (Translation between SMV-flat to mCRL2 "1-by-1")
Let SMV-flat model M_f, process specification P, the summand, the process-call expression, the set of initial values, the number of variables N and the *det* (determiniser) function be defined and denoted as in the definition of the SMV-flat to mCRL2 All-in-1 translation (Def. 73). Additionally, let the Counter Variable be denoted Cv (Def. 79), let Ei_x and En_x resp. be the init and next state valuation expression for variable V_x described in resp. Isf and Xsf. Also define the set of states in Vs_f denoted $V_1, V_2, ..., V_N$ to be the current state variables and let the set $Vs_{fe} = V_{1e}, V_{2e}, ..., V_{Ne}$ be the set of variables in which we temporarily record the already calculated next state valuations, with $Vs_{fe} = \{\langle N_f + \text{"\_}e'', T_f\rangle \mid \exists \langle N_f, T_f\rangle \in Vs_f\}$.

The translation from an SMV-flat to an mCRL2 1-by-1 model uses the same general SMV-flat to mCRL2 translations as is explained in the translation from SMV-flat to mCRL2 All-in-1. The 1-by-1 translation then translates SMV-flat model M_f to process specification P as follows:

- $N_b =$ "P".
- $Vs_b = Vs_f \cup Vs_{fe} \cup If \cup Cv$.
- $Sums_b =$
  $\{\langle If == True\ \&\ Cv == 0, \text{"}noAction\text{"}, pcall\rangle \mid \exists pcall \in \{\{V_{1e} = E \mid E \in det(Ei_1)\} \times Cv = 1\}$
  $\cup\ \{\langle If == True\ \&\ Cv == 1, \text{"}noAction\text{"}, pcall\rangle \mid \exists pcall \in \{\{V_{2e} = E \mid E \in det(Ei_2)\} \times Cv = 2\}$
  $\cup\ ...$
  $\cup\ \{\langle If == True\ \&\ Cv == N\text{-}1, \text{"}noAction\text{"}, pcall\rangle \mid \exists pcall \in \{\{V_{Ne} = E \mid E \in det(Ei_N)\} \times Cv = N\}$

  $\cup\ \{\langle If == False\ \&\ Cv == 0, \text{"}noAction\text{"}, pcall\rangle \mid \exists pcall \in \{\{V_{1e} = E \mid E \in det(En_1)\} \times Cv = 1\}$
  $\cup\ \{\langle If == False\ \&\ Cv == 1, \text{"}noAction\text{"}, pcall\rangle \mid \exists pcall \in \{\{V_{2e} = E \mid E \in det(En_2)\} \times Cv = 2\}$
  $\cup\ ...$
  $\cup\ \{\langle If == False\ \&\ Cv == N\text{-}1, \text{"}noAction\text{"}, pcall\rangle \mid \exists pcall \in \{\{V_{Ne} = E \mid E \in det(En_N)\} \times Cv = N\}$

  $\cup\ \{\langle Cv == N, \text{"}noAction\text{"}, pcall\rangle \mid \exists pcall \in \{V_1 = V_{1e}, V_2 = V_{2e}, ..., V_N = V_{Ne}, If = \text{"}False\text{"}, Cv = 0\}$

The **mCRL2 action specification** must be created to declare the "noAction" action. The **mCRL2 init statement** must be created with a random (but in range) values for each variable except for the initial flag which must be set to "True" and the counter variable which must be set to "0". The mCRL2 init statement should also state that the "noAction" action is allowed to be executed.

2.12 SMV-flat to mCRL2 "1-by-1" equivalence

An SMV-flat model and its translated mCRL2 1-by-1 model are equivalent up to the point that the 1-by-1 model takes multiple transitions to come to the same effects as the SMV-flat model does in a single transition. The reachable states are however the same if we do not consider states in the 1-by-1 model that are used for "in between" calculations. Such states are called internal states and are defined next.

Definition 81 (Internal state)
An internal state is a state in a Transition System which has been marked as internal. The intuition behind an internal state is that it should not be taken into consideration when equivalence of two reachable state spaces is determined, for example because they are part of in-between computation steps.

As such, it is important to determine which states exactly are marked as internal in the 1-by-1 models. In SMV-flat models there are no internal states.

Definition 82 (Internal states in the "1-by-1" model)
In any 1-by-1 model, only the states where the special **counter variable** is set to 0 are normal states. All other states are marked as internal as they are used to compute those normal states in a step by step (variable by variable) manner, or to copy calculated results to the current state variables.

Normal SMV-flat states and normal 1-by-1 states are not completely equal: The translation adds a copy of all SMV-flat variables denoted $v1_{fe}$ for variable $v1_f$, $v2_{fe}$ for variable $v2_f$ and so forth. The translation also adds the initial flag variable and the counter variable. For the initial flag and the counter variable we already know that their value bears no significance when we want to compare normal SMV-flat with normal 1-by-1 states as they will always be equal to "False" and "0" respectively. The copies of the state variables are also not important as they are used in order to store in-between computation results. As such we define what it means for two states to be equivalent in a state equivalence function called $stateEq_{1-by-1}$

Definition 83 (State equivalence relation for SMV-flat to 1-by-1)
Let M_f be an SMV-flat model and $TS_f = \langle S_f, I_f, T_f \rangle$ the induced Transition System. Let M_f have N_f state variables. Let M_b be an All-in-1 model and $TS_b = \langle S_b, I_b, T_b \rangle$ the induced Transition System. Let If denote the Initial Flag variable with valuation v_{if} and let Cv denote the Counter Variable with valuation v_{cv}. Let $s_f \in S_f$ and $s_b \in S_b$, then states s_f and s_b are equivalent if $(s_f, s_b) \in stateEq_{1-by-1}$. The state equivalence relation $stateEq_{1-by-1} \subseteq S_f \times S_b$ is defined as follows:

$$stateEq_{1-by-1} = \{((v1_i, v2_i, ..., vN_i), (v1_f, v2_f, ..., vN_f, v1_{fe}, v2_{fe}, ..., vN_{fe}, v_{if}, v_{cv})) \mid v1_i = v1_f, v2_i = v2_f, ..., vN_i = vN_f\}$$

Even though we only want to consider non-internal or "normal" states when comparing an SMV-flat model to a translated 1-by-1 model, it must still be the case that in the end the same normal states are reachable in both models. In order to help define that the set of reachable states must still be the same we first determine what an internal path is, after which we will explain the equivalence called weak bisimulation.

Definition 84 (Internal path)
An internal path, denoted $\Rightarrow$ is a path that starts and ends with a normal state. All other states on the path must be internal states.

Definition 85 (Weak Bisimulation Relation)
Given two Transition Systems TS and TS', a relation $R \subseteq S \times S'$ is a weak bisimulation relation if $(s, s') \in R$ implies

- if $s \Rightarrow t \in T$ then $\exists t' \in S'.s' \Rightarrow t' \in T' \wedge (t, t') \in R$.
- if $s' \Rightarrow t' \in T'$ then $\exists t \in S.s \Rightarrow t \in T \wedge (t, t') \in R$

Definition 86 (Weak bisimilar)
Two Transition Systems $TS = \langle S, I, T \rangle$ and $TS' = \langle S', I', T' \rangle$ are weak bisimilar if there exists a weak bisimulation relation R among then such that:

$$\forall s_0 \in I \ \exists s'_0 \in I' \ . \ (s_0.s'_0) \in R$$
$$\forall s'_0 \in I' \ \exists s_0 \in I \ . \ (s_0, s'_0) \in R$$

For the 1-by-1 model we have the same problem as with the All-in-1 model with the unique mCRL2 initial states. As such we will first attempt to proof that the 1-by-1 model generates a set of initial states that is equal to the set of SMV-flat initial states. Afterwards, assuming that the correct set of initial states

is generated, we will proof that both models are weak bisimilar. The definition of correct initial states generation does however needs to be adapted slichtly in order to incorporate the use of internal paths:

Definition 87 (Weak correct initial states generation)
Let the unique 1-by-1 initial state be denoted s_{0b}. An mCRL2 1-by-1 model with induced Transition System TS_b correctly generates a set of initial states equal to the set of initial states of an SMV-flat model with induced Transition System TS_f if:

- $\forall s_{0f} \in I_f \; \exists s \in S_b \; . \; s_{0f} \Rightarrow s \in T_f \wedge f(s_{0f}) = s$
- $\forall s \in S_b \mid s_{0b} \Rightarrow s \in T_b \; \exists s_{0f} \in I_f \wedge f(s_{0f}) = s$

2.13 SMV-flat to mCRL2 "1-by-1" proof

This section presents the proof that the SMV-flat to mCRL2 1-by-1 translation results in an 1-by-1 model that is **weak bisimilar** to the original SMV-flat model under the assumption an mCRL2 model is allowed to have multiple initial states which are equal to the SMV-flat initial states. In order to prove that assumption correct, we will first prove that the conditions for weak correct initial states generation are met. Afterwards we will prove that together with the assumption that the initial states are correctly generated, the two models are weak bisimilar under the weak bisimulation relation $stateEq_{1-by-1}$.

Theorem 88
Let M_f be an SMV-flat model and let M_b be a 1-by-1 model that is obtained from M_f by applying the SMV-flat to mCRL2 1-by-1 translation. Let $TS_f = \langle S_f, I_f, T_f \rangle$ and $TS_b = \langle S_b, I_b, T_b \rangle$ be resp. the Transition System induced by M_f and M_b. Then the following must hold:

TS_f and TS_b are weak bisimilar with weak bisimulation relation $stateEq_{1-by-1}$. The weak bisimulation holds under the assumption that mCRL2 models are allowed to have multiple initial states, which have to be proved equal to the initial states of the SMV-flat model.

Proof

In order to be able to prove anything we first have to confirm that we can use function $stateEq_{1-by-1}$. This function assumes that if the state vector in an SMV-flat model has N_f variables then the translated All-in-1 model has $(N_f * 2) + 2$ variables, for which the first N_f are have equal types as the set Vs_f, just as the second N_f variables. The last two variables must be equal to the Initial Flag variable and the Counter Variable.

Let $M_f = \langle Vsf, Dsf, Isf, Xsf \rangle$. Let the set of states in Vs_f, denoted $Vf_1, Vf_2, ..., Vf_N$, be the current state variables and let the set $Vs_{fe} = Vf_{1e}, Vf_{2e}, ..., Vf_{Ne}$ be the set of next state variables, with $Vs_{fe} = \{\langle N_f + \text{"\_}e'', T_f \rangle \mid \exists \langle N_f, T_f \rangle \in Vs_f\}$. Let the state vector for states in TS_f be denoted as $SV_f = \langle Vf_1, ..., Vf_{N_f} \rangle$. Let Ei_x and En_x be resp. the init and next state valuation expression for variable Vf_x described in resp. Isf and Xsf. Let the mCRL2 process specification of the M_b model be denoted $P = \langle N_b, Vs_b, Sums_b \rangle$. Let the state vector for model M_b be denoted SV_b.

Proof of: $SV_b = \langle Vf_1, ..., Vf_{Nf}, Vf_{1e}, ..., Vf_{Nfe}, If, Cv \rangle$. State vectors consist of all state variables in a model. For M_f those are the variables in Vsf and for M_b those are the variable in Vs_b. The SMV-flat to mCRL2 1-by-1 translation states that $Vs_b = Vs_f \cup Vs_{fe} \cup If \cup Cv$ which directly proofs the correct contents of SV_b.

Let's now take a look at the conditions for correct initial states generation as defined in Def. 87.

Proof of: $\forall s_{0f} \in I_f \ \exists s \in S_b \ . \ s_{0f} \Rightarrow s \in T_f \wedge stateEq_{1-by-1}(s_{0f}) = s$ and $\forall s \in S_b \mid s_{0b} \Rightarrow s \in T_b \ \exists s_{0f} \in I_f \wedge stateEq_{1-by-1}(s_{0f}) = s$

Following the same reasoning as used in the proof for the SMV-flat to All-in-1 translation, we see that the SMV-flat to mCRL2 1-by-1 translation creates an mCRL2 init statement with random values for each variable except for variable If which is set to "$True$", and variable Cv which is set to "0". The random values combined with the If and Cv values are therefore exactly the contents of state s_{0b}. Starting with this state, all summands that may generate next states therefore have to have a condition that states that $If ==$ "$True''$ and $Cv == 0$.

$$\{\langle If = \mathit{True} \ \& \ Cv = 0, \text{"}noAction\text{"}, pcall \rangle \mid \exists pcall \in \{\{V_{1e} = E \mid E \in det(Ei_1)\} \times Cv = 1\}$$

The states generated by these summands all have $Cv == 1$ and $If ==$ "$True''$ (as non-mentioned variables in a process-call stay the same). The states also have a calculated next state value for variable V_1 recorded in variable V_{1e}. As $Cv \neq 0$, we see that the generated states are internal states. As such we have not yet found a normal state and are on the way of defining an internal path. Knowing that the generated states have $Cv == 1$ and $If ==$ "$True''$, we can only create new next state by applying the following summands:

$$\{\langle If == \mathit{True} \ \& \ Cv == 1, \text{"}noAction\text{"}, pcall \rangle \mid \exists pcall \in \{\{V_{2e} = E \mid E \in det(Ei_2)\} \times Cv = 2\}$$

We now end up in internal states that have the first two variables calculated which is reflected in their Counter Variable. Continuing to generate states like this, we at some point will generate states in which all variable's next value have been calculated and in which $Cv == N_f$. At that moment the copy transition becomes active as it is the only summand that has the correct condition:

$$\{\langle Cv == N, \text{"}noAction\text{"}, pcall \rangle \mid \exists pcall \in \{V_1 = V_{1e}, V_2 = V_{2e}, ..., V_N = V_{Ne}, If = \text{"}False'', Cv = 0\}$$

After all next state value's have been copied to the current state variables the copy transition makes sure that the Initial Flag is set to false and that the Counter Variable is reset to 0, creating a non-internal, or also called "normal" state. Therefore, the internal path ends and a state has been found that is (hopefully) equal to an SMV-flat initial state.

We will now prove that the normal states generated by the above process results in a set of mCRL2 states that is equal (according to $stateEq_{1-by-1}$) to the set of SMV-flat initial states.

The SMV-flat initial states are generated by the expression $I_f = i_1 \times i_2, \times ... \times i_n$, with the sets of possible initial values per variable denoted as i_1 for variable V_1, i_2 for variable V_2 and so forth. In the All-in-1 translation proof we already argued that $i_1 == det(Ei_{1f})$, $i_2 == det(Ei_{2f})$ and so forth. The summands created by the SMV-flat to mCRL2 1-by-1 translation that have the condition $If ==$ "$True$" $\& \ Cv == 0$ effectively calculate the next state valuation for the first variable, which can be denoted as

$V_1 = E \mid E \in det(Ei_1)$, knowing that later on the copy transition happens. These summands result in a set of states that all have one of the possible values in $det(Ei_1)$. After creating a new set of next states from the summands that have the condition $If ==$ "$True$" & $Cv == 1$, we have effectively computed all states that have some value from $det(Ei_1)$ for the first variable and some value from $det(Ei_2)$ for the second variable. All states in the product of those two sets have been computed: $det(Ei_1) \times det(Ei_2)$. After repeating this process a number of times, states appear that have variable values $If ==$ "$True$" & $Cv == N - 1$. After computing their next states we have effectively computed $det(Ei_1) \times det(Ei_2) \times ... \times det(Ei_N)$. As $i_1 == det(Ei_{1f})$, $i_2 == det(Ei_{2f})$ and so forth, and as the $stateEq_{1-by-1}$ drops all extra duplicate variables and the If and Cv variables it becomes clear that the computed normal states, via a path of only internal states, must be equal (according to $stateEq_{1-by-1}$) to the set of SMV-flat initial states.

As such, we have proven both $\forall s_{0f} \in I_f \; \exists s \in S_b . s_{0f} \Rightarrow s \in T_f \wedge stateEq_{1-by-1}(s_{0f}) = s$ and $\forall s \in S_b \mid s_{0b} \Rightarrow s \in T_b \; \exists s_{0f} \in I_f \wedge stateEq_{1-by-1}(s_{0f}) = s$.

As the definition of weak correct initial state generation implies all conditions on the initial states of the weak bisimulation relation definition we will skip those and move on to the conditions on any related states by $stateEq_{1-by-1}$.

Proof of: If $s \Rightarrow t \in T$ then $\exists t' \in S' . s' \Rightarrow t' \in T' \wedge (t, t') \in stateEq_{1-by-1}$ and $s' \Rightarrow t' \in T'$ then $\exists t \in S$. $s \Rightarrow t \in T \wedge (t, t') \in stateEq_{1-by-1}$.
We know that state t is a "normal" state that is not the mCRL2 unique initial state and as such $If ==$ "$False''$ & $Cv == 0$ in state t. Using the same reasoning as for the proof of Weak correct initial state generation, but this time using the next state valuation expressions (En_x instead of Ei_x for each variable), we see that the 1-by-1 summands effectively calculates a set of normal states whose next state valuation is in $det(En_1) \times det(En_2) \times ... \times det(En_N)$, applied by the process-call to state t.

By the reasoning in the proof of the All-in-1 translation we know that the SMV-flat model calculates next states with the following function:

$$T_f = \{(s_1, s_2) \mid s_2 \in n_1(s_1) \times n_2(s_1) \times ... \times n_N(s_1)\}$$

That same proof showed us that $det(En_1) == n_1(s_1)$, $det(En_2) == n_2(s_2)$ etcetera. As $(s, t) \in stateEq_{1-by-1}$, and as their next state calculation is shown to be equal we have proven that if we only look at "normal" state in the 1-by-1 model connect to each other by internal paths, then the state spaces of the All-in-1 model and the 1-by-1 model are exactly the same. As such, any two states that are related by $stateEq_{1-by-1}$ can simulate each other's behaviours as required by the definition of the Weak Bisimulation Relation.

2.14 Dependency matrix, state count and textual size of the created models

This Section investigates a number of practical details on the created translations and also suggests the creation of two additional translations to fix some of the uncovered practical problems. Per translation, we will investigate the contents of the LTSmin dependency matrix, the number of reachable states in relation to the number of reachable states in the original SMV-flat model and the textual size of the model. The dependency matrix and the number of reachable states are important for the efficiency of the reachability analysis by the LTSmin tool. The size of the mCRL2 model is important as the translation tool that generates the mCRL2 model, and the mcrl22lps pre-processing tool might get into trouble when the textual size gets too big.

Before we look at each of the translations lets first determine what the dependency matrix looks like for an mCRL2 model. The dependency matrix has on one side a number of groups and on the other side the state variables of the mCRL2 model. Each group represents a number of transitions which in the case of an mCRL2 model are those that are generated by a single mCRL2 summand. The information in the matrix captures whether a group depends on a certain variables. LTSmin strongly prefers models that have as less dependencies as possible as that enables an efficient reachability search. The mechanics in the LTSmin toolset that use the dependency matrix in order to do reachability analysis are described in Section 1.5.1.

2.14.1 Details for All-in-1 models

The All-in-1 models have a very bad dependency matrix as each summand changes all variables at the same time (because all process-call expressions change all variables). Therefore all created groups depend on all state variables. As discussed before, the state space of the All-in-1 translation is the same size plus one state as the SMV-flat state space. This is due to the addition of the unique mCRL2 initial state. All-in-1 models are generally large when talking about textual size. This is due to the fact that the All-in-1 translation creates a summand for each possible combination of the results of the available non-deterministic choices. In each such summand it generates the next expressions for all variables. As such the size of a single summand is fairly big, and the number of summands gets exponentially larger as linearly more non-deterministic expressions are available or the existing non-deterministic expressions have linearly more choices.

2.14.2 Details for 1-by-1 models

The concept of the 1-by-1 model was created in order to generate equivalent mCRL2 models with a better dependency matrix than the All-in-1 model has. In order to generate mCRl2 models with a better dependency matrix the 1-by-1 translation creates summand that only change a single state variable at the same time. As such it does introduce a significant number of extra states. 1-by-1 models also have double the number of variables as All-in-1 models. The extra variables are needed to store calculated next state values while the old current state values may also still be needed and as such cannot yet be thrown away. The SMV-flat to mCRL2 1-by-1 translation was also created to create mCRL2 files that where of a smaller size than their equivalent All-in-1 models. This was mostly of importance as the translation tool had great problems with generating the big All-in-1 models.

1-by-1 dependency matrix

A 1-by-1 model has a dependency matrix has twice the variables compared to the dependency matrix of an All-in-1 model because the extra set of next state variables are added and the counter variable is added to the set of state variables. Each summand in a 1-by-1 model is generally concerned with calculating the value of a single variable (denoted V, with next state variable V_e). To that end it writes to two variables: V_e and the special Counter Variable. The summand also reads the current state variable for each variable that is needed to calculate the next state value of variable V. For these summands the dependency matrix therefore reflects very real and necessary dependency relations.

The 1-by-1 model also has a special summand that copies all next state variables to the current state variables once the next state values have been determined for all state variables. This summand therefore represents a big flaw in all 1-by-1 models: The copy summand depends on all variables as it reads form all next state variables and writes to all current state variables. It also resets the Counter Variable and sets the Initial Flag variable to "$False$". As such it depends on all variables in some way and therefore the summand's corresponding group in the dependency matrix depends on all variables.

1-by-1 state calculation

The 1-by-1 model obtains a better dependency matrix in exchange for a larger set of reachable states. The exact number of the blow-up in reachable states strongly depends on the number of non-deterministic expressions in the original SMV-flat model and the number of non-deterministic options that each such an expression has. Figure 2.2 in the preview Section gives an idea of how to calculate the reachable state space blow-up. We will calculate an estimate based on the number of summands that are used to calculate the next state of any given mCRL2 state. Let the SMV-flat model have N state variables. Starting from any mCRL2 state that is equal to a state in the SMV-flat state space, we know that the Counter variable Cv equals 0. The enabled summands calculate the next state value of the first variable V_1. Denoting the next expression that belongs to V_1, we know that the amount of enabled summands must be equal do $det(E_1)$. We now end up with say L_1 next states, which have a counter variable that is equal to 1. Generating the states from the states in L_1 thus happens by looking at the summands that have $Cv = 1$ as their condition. The enabled summands are those that calculate the next state value for variable V_2. The amount of enabled summands is $det(E_2)$. By that reasoning, the calculation of the number of states after all variables have been analysed is:

$$
\begin{aligned}
L_1 &= |det(E_1)| \\
L_2 &= L_1 + (L_1 * |det(E_2)|) \\
L_3 &= L_2 + (L_2 * |det(E_3)|) \\
&... \\
L_N &= L_{N-1} + (L_{N-1} * |det(E_N)|)
\end{aligned}
$$

The same calculation can be made for the number of states that are needed to calculate the set of initial states by using the initial expressions of the variables instead of the next expressions. We do not count the states created by the copy summand because those states are again equal to a state in SMV-flat and therefore should not be counted when calculating the state blow-up in regards to the number of states reachable in the SMV-flat model.

Note that the calculated value is only an indication of the state space blow-up as the calculation is based on the number of executed summands. Therefore, the calculation does not take the possibility into account that some summands may results in the same state. The calculated estimate does however give an upper bound on the reachable state space blow-up.

1-by-1 textual size

1-by-1 models are of significant smaller size then there All-in-1 counterpart. Each summand only contains the next expression for a single state variable and summands are only created for all possible combinations of non-deterministic choices of a single variable, and not for all variables at the same time. Therefore 1-by-1 model files are relatively small in size.

2.14.3 1-by-1 Sum

The 1-by-1 Sum translation is an not before mentioned variant of the 1-by-1 translation that groups mCRL2 summands together that have equal dependencies on all variables. When creating summands to update variable V_x with next expression E_x, the original SMV-flat to mCRL2 1-by-1 translation creates a summand for each combination of non-deterministic choices in $det(E_x)$. The individual groups resulting from these transitions don't have a real function however, because all summands assign values to V_{xe}, to Cv and read from all variables that are mentioned in E_x. The groups therefore have identical dependency characteristics and can therefore be grouped. Grouping is arranged by creating a single summand that has mCRL2 Sum expressions:

Definition 89 (mCRL2 Sum expression)
An mCRL2 Sum expression consists of an **mCRL2 symbolic constant** and an **mCRL2 type**. The sum expression is a language construct that allows to combine the description of multiple summands into a single one. It does so by allowing to use the symbolic constant as a place holder variable in the summand that the Sum expression is created in. In the induced Transition System the summand that contains the Sum expression is turned back into a set of summands, that have instantiations of each Sum expression's symbolic constant. The available instantiations are determined by the Sum expression's type. Multiple sum expressions result in summands that have instantiations of the symbolic constants from the Cartesian product of all the sum expression's types.

A 1-by-1 Sum model therefore has two dependency matrix groups for each variable (one for $If ==$ "$False$" and one for $If ==$ "$True$"). The dependency matrix also contains one group for the copy transition. In any other aspect the dependency matrix is the same as for the original 1-by-1 model. A 1-by-1 Sum model is also shorter in textual size as summands are grouped together. The reachable state space of a 1-by-1 Sum model is exactly the same as the original 1-by-1 model as the summands are grouped together, but effectively expanded again when calculating the set of possible next states.

2.14.4 1-by-1 Sum Copy

The 1-by-1 Sum Copy translation is a not before mentioned variable of the 1-by-1 Sum translation (but could just as well work on the original 1-by-1 translation). The 1-by-1 Sum Copy translation attempts to have an even better dependency matrix by splitting up the copy transition into multiple transitions that each copy a single variable. The copy transition is a problem in the 1-by-1 and 1-by-1 Sum dependency

matrix as it depends on all variables. Therefore the 1-by-1 Sum Copy again trades an even better dependency matrix for an increase in reachable states. The resulting dependency matrix is the same as the one described for the 1-by-1 translation with the exception that the on all variables depending copy summand group is now replaced with multiple groups that all depend on only 3 variables: the current and next state version of the variable that is being copied plus the counter variable.

In comparison with the 1-by-1 Sum translation the textual size of the model increases slightly (but only linearly in the number of states). The calculation of the number of reachable states can be given by the following calculations:

$L_1 = |det(E_1)|$
$L_2 = L_1 + (L_1 * |det(E_2)|)$
$L_3 = L_2 + (L_2 * |det(E_3)|)$
...
$L_N = L_{N-1} + (L_{N-1} * |det(E_N)|)$
$L_{C1} = L_N + L_N$
$L_{C2} = L_{C1} + L_N$
...
$L_{CN} = L_{CN-1} + L_N$

This calculation assumes that the last copy transition is followed by a transition that changes the Initial Flag to "$False$" and the Counter Variable to 0. The states generated by that transition are not counted as they are again "normal" states. This calculation can also be done for the number of states that are needed to calculate the set of initial states. As with the original 1-by-1 states calculation the results of the calculation only represent an upper bound on the actual blow-up in the number of states relative to the number of reachable states in the SMV-flat model.

2.14.5 Comparison between the translations

Table 2.5 gives a comparison of the presented translations in the categories of the dependency matrix, the reachable state spaces and the textual size of the produced models. The comparison is done on the "Worst, Worse, Average, Better, Best" scale.

Table 2.5: A comparison between the translations

	Dependency matrix	Reachable state space	Textual size
All-in-1	Worst	Best	Worst
1-by-1	Average	Worse	Better
1-by-1 Sum	Average	Worse	Best
1-by-1 Sum Copy	Best	Worst	Best

Chapter 3

Results

This chapter describes the results of the automatic translations and the execution times of the original models versus their translated counterparts. The chapter also describes results of some other comparison methods, such as profiling results of the NuSMV and LTSmin symbolic reachability tools. All experiments are performed on a visualised Ubuntu 11.10 64bit environment, with 4 CPU cores and 4GB memory. The virtualization software used was VMware workstation version 9.0.1, running on a system with Windows 8.0. The native windows computer has an Intel Q9550 2,83Ghz quad core CPU, 8GB of memory and an OCZ Vertex 3 SSD 240GB.

For the experiments NuSMV version 2.5.4 was used, always in conjunction with the "-r" option. This option ensures that NuSMV will perform the reachability analysis. Before running any files in NuSMV all specifications where removed from the SMV models in order to ensure that the validity of any specifications are not calculated as well. For LTSmin we used LTSmin version 2.0. In order to be able to run mCRL2 models in LTSmin the mcrl22lps tool from the mCRL2 toolset must first be executed. The version of the used mCRL2 toolset was 201202.0. The LTSmin tool used for the actual reachability is called lps2lts-sym, which just as the mcrl22lps tool where run with multiple options as enumerated below. Only the best results of the multiple runs with different enabled options where used when the results were collected. All results where measured three times after which the average was taken.

- no-cluster: Uses the option –no-cluster on mcrl22lps.
- advanced: Uses the option –no-cluster, –no-globvars and –delta on mcrl22lps. Uses -rgs on lts2lps-sym.
- chain: Uses the option –no-cluster, –no-globvars and –delta on mcrl22lps. Uses -rgs and –order=chain-prev on lts2lps-sym.
- sat-like: Uses the option –no-cluster on mcrl22lps. Uses –saturation=sat-like on lts2lps-sym.
- save-sat: Uses the option –no-cluster, –no-globvars and –delta on mcrl22lps. Uses -rgs, –saturation=sat-like and –save-sat-levels on lts2lps-sym.

For all experiments we used the translation from SMV define macros to mCRL2 mappings (Definition 70). The alternative was to use Definition 69 which proposes to substitute the define variable's next expressions in all places where they are used. Tests showed that either option did not result in different performance results. We have chosen to use the mCRL2 mappings as the substitution solution has the disadvantage

that it may give rise to outputting the define variable's next expression multiple times which blows up the size of the model file and decreases it's readability.

3.1 Used models

The models used in the experiments are described in this section. We shortly discuss the characteristics, the meaning and the origin of the used models. Tables 3.1 and 3.2 contain some characteristics of the used models. The non-determinism row of the tables denotes how much non deterministic choices are made when determining a next state of the model. The used format shows on each numeric position a non-deterministic choice and the number is actual number of non-deterministic options for that choice to choose from. The meaning of the models and their special characteristics are discussed afterwards on a per model basis.

Table 3.1: Model characteristics

	periodic	Tictactoe_v2	door	ferryman	abp4
Model type	synchronous	synchronous	synchronous	synchronous	asynchronous
Modules (sync - async - main)	0-9-1	0-0-1	0-0-1	0-0-1	4-4-1
Variables (state - define)	11-29	15-1	4-0	6-8	12-0
States (NuSMV)	1k, 100, 10k	887k	36	240	140k
Non-determinism	10,0,10x10	3x3x3x3	2x2x3x5	8x8	16
Versions	orig, corr, extra				abp4

Table 3.2: Model characteristics

	filo_n	production-cell	screen.1001	screen.107	screen.125
Model type	asynchronous	synchronous	synchronous	synchronous	synchronous
Modules (sync - async - main)	n-n-1	0-9-1	0-0-1	0-0-1	0-0-1
Variables (state - define)	n*2 + 1	39-0	26	23	21
States (NuSMV)	40, 218	81	510k	41k	11k
Non-determinism	2x2x2*n	none	4	4	4
Versions	filo3, filo4, filo5				

periodic -orig, -corr and -extra

The periodic model was found in the NuSMV example set [1]. It is a data driven pipeline synchronous SMV model. The aux variable in the original model is not used for anything and annotated with the comment "used just to make SMV recognize the symbols idle and p*". We assume that the author did not yet have the fairly new CONSTANTS clause available when making the model as users can freely declare enumeration values in the CONSTANTS section without creating an extra state variable. As such, model periodic-corr removes the aux variable and adds an equivalent CONSTANTS clause. Periodic-extra adds an extra aux variable called aux2 in order to see how the translation and execution of the created models react to more

[1] http://nusmv.fbk.eu/examples/examples.html

nondeterminism. Each aux variable is an input variable with 10 symbolic constants and therefore blows up the state space 10 fold.

Tictactoe_v2, Door and Ferryman

The tictactoe smv model is a model of the well known Tic Tac Toe game. In a 3x3 grid two players must take turns to place one of their personal tokens. Once a player has three in a row (either horizontal, vertical or diagonal) that player wins. We have two versions, the forSMV and the v2 version. As both versions has comparable results we have only shown version v2 in our results. The "door" SMV model represents a model of the logics behind an automatic door with open and close buttons on both sides of the door. This model is very small in state space (36 states) but still results in interesting results as it contains multiple non-deterministic choices. The ferryman SMV model is a model of the ferryman puzzle: A fisherman with his goat, cabbage and wolf stand on one side of the river. The fisherman want to cross the river on a boat that can only hold himself and another item. If the wolf and the goat are left alone on one side of the river then the wolf will eat the goat. If the goat and the cabbage are left alone then the goat will eat the cabbage. All "possessions" of the fisherman must reach the other side of the river safely. All three models where found in the example set of the NuSMV-tools Model Advisor [2].

filo_n and abp

The filo_n models are a SMV representation of the well known dining philosophers problem, with n philosophers and n forks. The problem is valid for n >= 2: n philosophers sit around a round table, with the space between philosophers occupied with a single fork. Each philosopher can either think or eat. When a philosopher wants to eat he must first acquire the two forks that are situated on both his sides. Seeing as there are as many forks as there are philosophers, the problem is to find a solution of how the philosophers should attempt to claim the forks in such a way that all philosophers regularly get to eat (no philosophers should starve). The filo models where found in the example set of the NuSMV-tools Model Advisor. The abp models are descriptions of the Alternating Bit Protocol which is a communication protocol. A sender and a receiver are connected by two communication channels which together serve as a bi direction communication medium. Both the sender and the receiver have a binary alternating bit which is used in order to determine if messages are correctly received. A detailed description of the Alternating Bit Protocol can be found in the apbx.smv files. The abp models are part of the NuSMV examples set [3]. The used abp4C.smv model is a copy of the original abp4.smv model with some adjustments to the enumeration types as to keep symbolic constants unique.

production-cell

The production-cell model describes an ASM-specification of the Production Cell and is part of the NuSMV examples set. The model describes an automatic industrial system which involves a crane, a robot with two arms and multiple transport belts. See the paper by Börger and Mearelli for a more detailed description [15]. The used production-cellC.smv model is a copy of the original production-cell.smv model with some adjustments to the enumeration types as to keep symbolic constants unique.

[2] http://code.google.com/a/eclipselabs.org/p/nusmv-tools/
[3] http://nusmv.fbk.eu/examples/examples.html

screen.1001, screen.107 and screen.125

The screen files are Sokoban puzzles. These puzzles are translated to a SMV model by software created in the Modelling & Analysis of Concurrent Systems 2 course given at the University of Twente by Jaco van de Pol. A Sokoban puzzle looks like the listing below. W stands for wall, P for player, B for box and "." for the goal positions. The player can walk in all directions: up, down, left and right. The player cannot walk into a wall and if the player pushes against a box then the box will move in the direction the player is pushing it. This however cannot happen if the box is up against the wall or against another box. The goal of the puzzle is to control the player in such a way that all boxes are pushed onto a goal position.

```
1  WWWWW
2  WP  WWWWW
3  W WB .  W
4  W    .W W
5  WW BW.B W
6   WW   WWW
7    WWWWW
```

Used LTSmin options

Tables 3.3 and 3.4 shows the LTSmin options per translation that worked best for the presented models.

Table 3.3: LTSmin options per translation per model

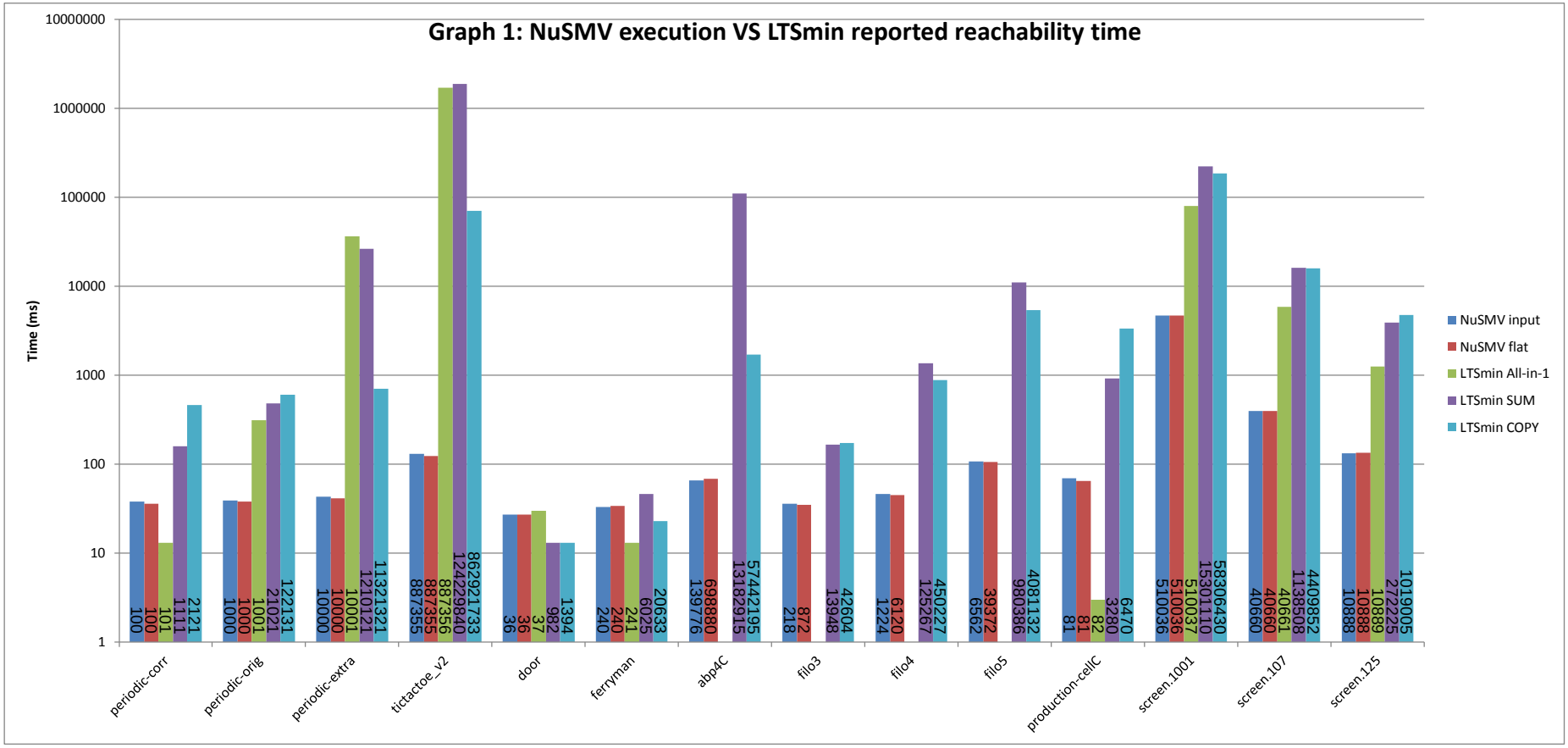
	p-orig	p-corr	p-extra	Tictactoe_v2	door	ferryman	abp4
All-in-1	chain	chain	chain	advanced	chain	chain	-
1-by-1 Sum	sat-like	sat-like	advanced	sat-like	advanced	no-cluster	sat-like
1-by-1 Sum Copy	chain	sat-like	sat-like	sat-like	no-cluster	sat-like	sat-like

Table 3.4: LTSmin options per translation per model

	filo-3, filo-5	filo-4	production-cell	screen.1001	screen.107	screen.125
All-in-1	-	-	chain	advanced	advanced	chain
1-by-1 Sum	sat-like	no-cluster	sat-like	sat-like	sat-like	sat-like
1-by-1 Sum Copy	sat-like	no-cluster	sat-like	sat-like	sat-like	sat-like

3.2 Measuring the LTSmin and NuSMV reachability tool execution times

A first comparison was made by measuring the full execution time of the NuSMV reachability tool against the reported reachability time of the LTSmin reachability tool. We chose not to measure the full execution time of the LTSmin tool as it contains a preprocessing step in which some mCRL2 related files are compiled. The experiment includes measurements for the original SMV model, the flattened SMV model, the translated mCRL2 All-in-1 model, the translated mCRL2 1-by-1 SUM model and the translated mCRL2 1-by-1 COPY model. We do not report on models resulting from the original mCRL2 1-by-1 translation as it has exactly the same performance as the mCRL2 1-by-1 SUM translation. The results are depicted in Graph 1 on the next page. The x-axis shows (created) models per input model. The y-axis shows the time in milliseconds. The numbers in the base of the vertical bars represent the number of states generated by the applied reachability tool. LTSmin All-in-1 measurements are not available for the abp4C, filo3, filo4 and filo5 models as the mcrl22lps tool ran out of memory. Please note that the y-axis uses a logarithmic scale base 10.


Graph 1: NuSMV execution VS LTSmin reported reachability time
Time (ms)
10000000
1000000
100000
10000
1000
100
10
1
NuSMV input
NuSMV flat
LTSmin All-in-1
LTSmin SUM
LTSmin COPY
periodic-corr
100 100 101 1111 2121
periodic-orig
1000 1000 1001 21021 122131
periodic-extra
10000 10000 10001 1210121 11321321
tictactoe_v2
887355 887355 887356 12422840 86292733
door
36 36 37 982 1394
ferryman
240 240 241 6025 20633
abp4C
139776 698880 1318295 57442195
filo3
218 872 13948 42604
filo4
1224 6120 125267 450227
filo5
6562 39372 980386 4081132
production-cellC
81 81 82 3280 6470
screen.1001
510036 510036 510037 1530110 5830643
screen.107
40660 40660 40661 1138508 4409852
screen.125
10888 10888 10889 272225 1019005

The data in Graph 1 shows that NuSMV is significantly faster in most cases in comparison to all LTSmin measurements. Differences of multiple orders of magnitude have been measured. The "door", "ferryman" and "periodic-cellC" models do not show a significant difference which is most likely due to their very small state spaces. Even though the All-in-1 models sometimes have good performance it is important to realize that due to the model's textual size the mCRL2 preprocessing steps generally take significantly longer. So much longer, that we were unable to take measurements for the abp4C and filox models as the mcrl22lps tool ran out of memory after a long runtime. The periodic-corr, periodic-orig and periodic-extra models show how the LTSmin reachability time is influenced upon an increase of non-deterministic choices in the model. The results show that the All-in-1 and 1-by-1 Sum model's execution times are strongly influenced by the presence of the extra non-deterministic choices in periodic-orig and periodic-extra whereas the 1-by-1 Copy and the NuSMV execution time are not.

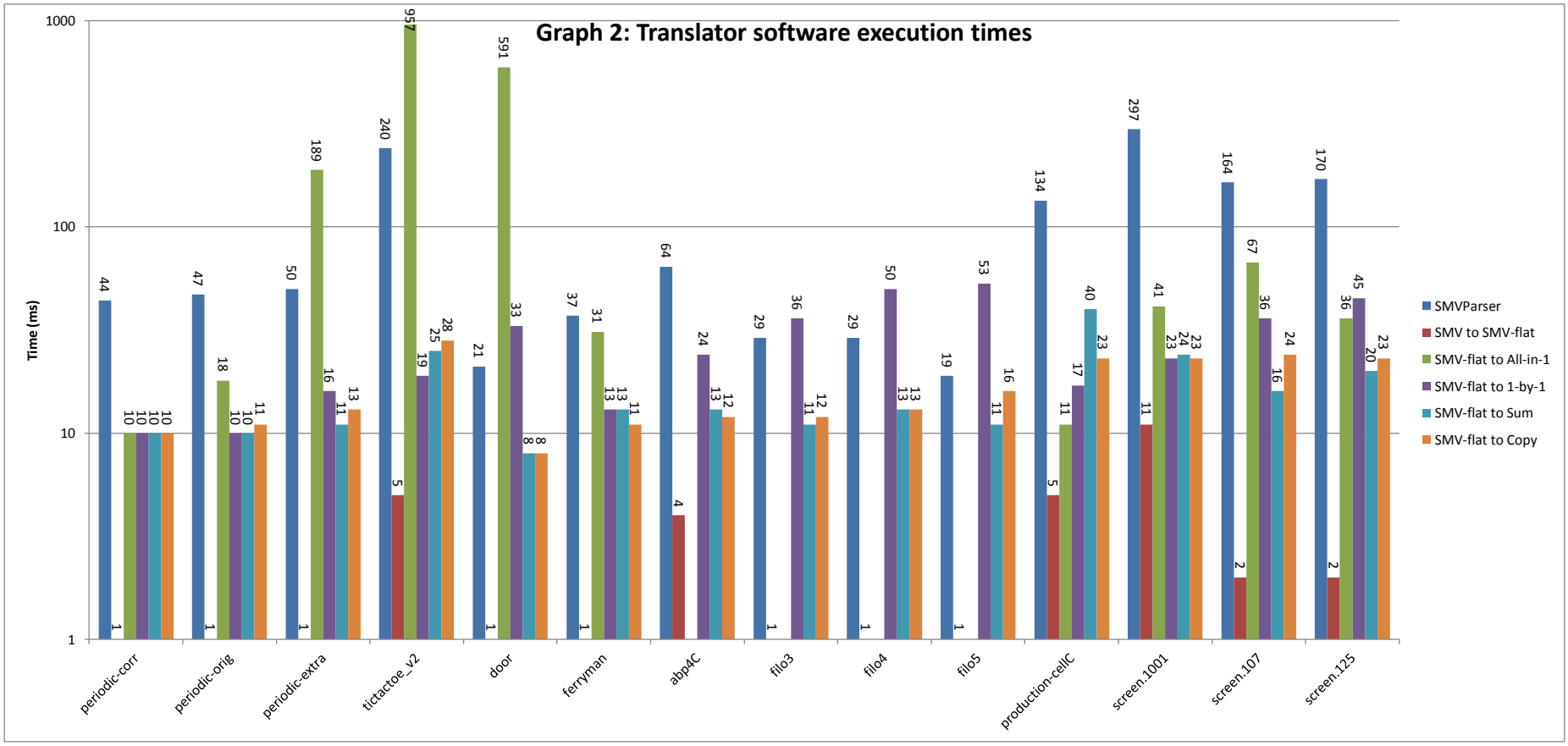
The abp4, filo3, filo4 and filo5 models are special as they are asynchronous models. Graph 1 shows that it does not seem to matter whether the input model is synchronous or asynchronous. This observation can be explained by the fact that the presence of multiple asynchronous processes is translated to an extra non-deterministic by the translation from SMV to SMV-flat. The synchronous models may however also contain non-deterministic choices and as such there is no real difference in the synchronous and asynchronous models after translation. The results for the abp4C model do however show a clear difference between the original 1-by-1 Sum and the 1-by-1 Copy versions of the model which must be an effect of the better dependency matrix resulting from the Copy version of the model. Any occurrences where a Copy model performs worse than the Sum model may be due to the increased number of dependency groups which might effect reachability negatively.

The results for the Sokoban models screen.1001, screen.107 and screen.125 show that these models present more of a challenge to the NuSMV reachability tool than the other models. It seems that the execution time difference between the NuSMV and LTSmin reachability tools stays roughly the same for all three models which suggests that this is also the case for even larger Sokoban puzzles. This observation stands against what we see with the periodic models that show an increase in the difference between NuSMV and LTSmin as we increase the number of non-deterministic choices. The difference can be explained with the fact that all Sokoban puzzle models only have a single non-deterministic choice with four options that determines the direction of the player.

Graph 1 also shows that the SMV-flat to All-in-1 translation is the best translation in the sense that the resulting model is evaluated faster than the models created by the 1-by-1 translations. This is most likely explained by the larger number of states in the state space of the 1-by-1 models. The big disadvantage of the All-in-1 translation is that models tend to get very big, resulting in long execution times from the translation tool and long execution times (and high memory usage) of the mcrl22lps tool.

3.3 Measuring the execution time of the translator tool

While measuring all data shown in Graph 1 we also measured the time it took for our translation software to generate the desired models. The measurements are shown in Graph 2 (on the next page). It is important to measure the execution time of the translation software in order to determine its practical use for end users: The translation is only useful if the time it takes to translate the model plus the reachability time of the new model is smaller than the original reachability time.

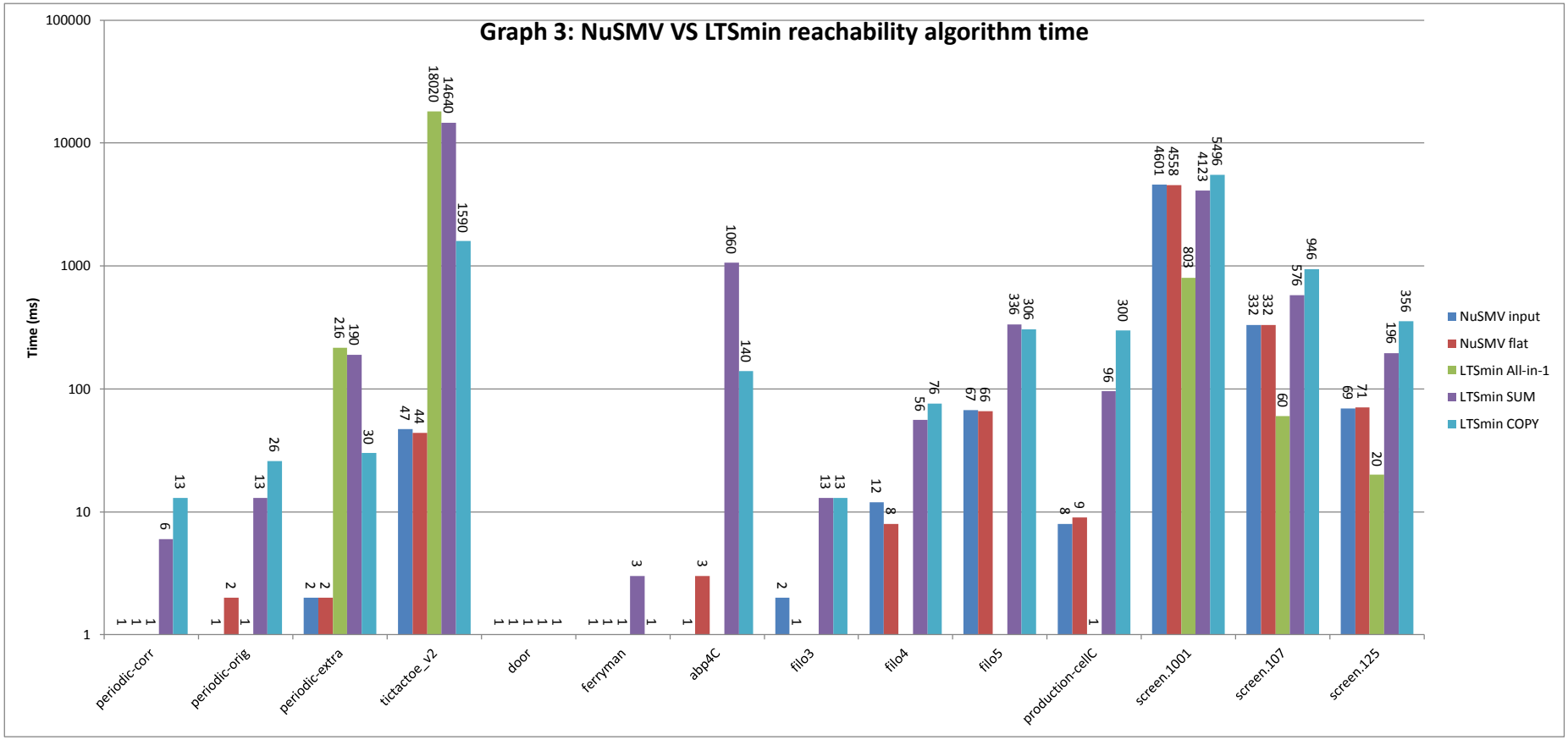

Graph 2: Translator software execution times
Time (ms)
1000
100
10
1
SMVParser
SMV to SMV-flat
SMV-flat to All-in-1
SMV-flat to 1-by-1
SMV-flat to Sum
SMV-flat to Copy
periodic-corr
periodic-orig
periodic-extra
tictactoe_v2
door
ferryman
abp4C
filo3
filo4
filo5
production-cellC
screen.1001
screen.107
screen.125
44 1 10 10 10 10
47 1 18 10 10 11
50 1 189 16 11 13
240 5 957 19 25 28
21 1 591 33 8 8
37 1 31 13 13 11
64 4 24 13 12
29 1 36 11 12
29 1 50 13 13
19 1 53 11 16
134 5 11 17 40 23
297 11 41 23 24 23
164 2 67 36 16 24
170 2 36 45 20 23

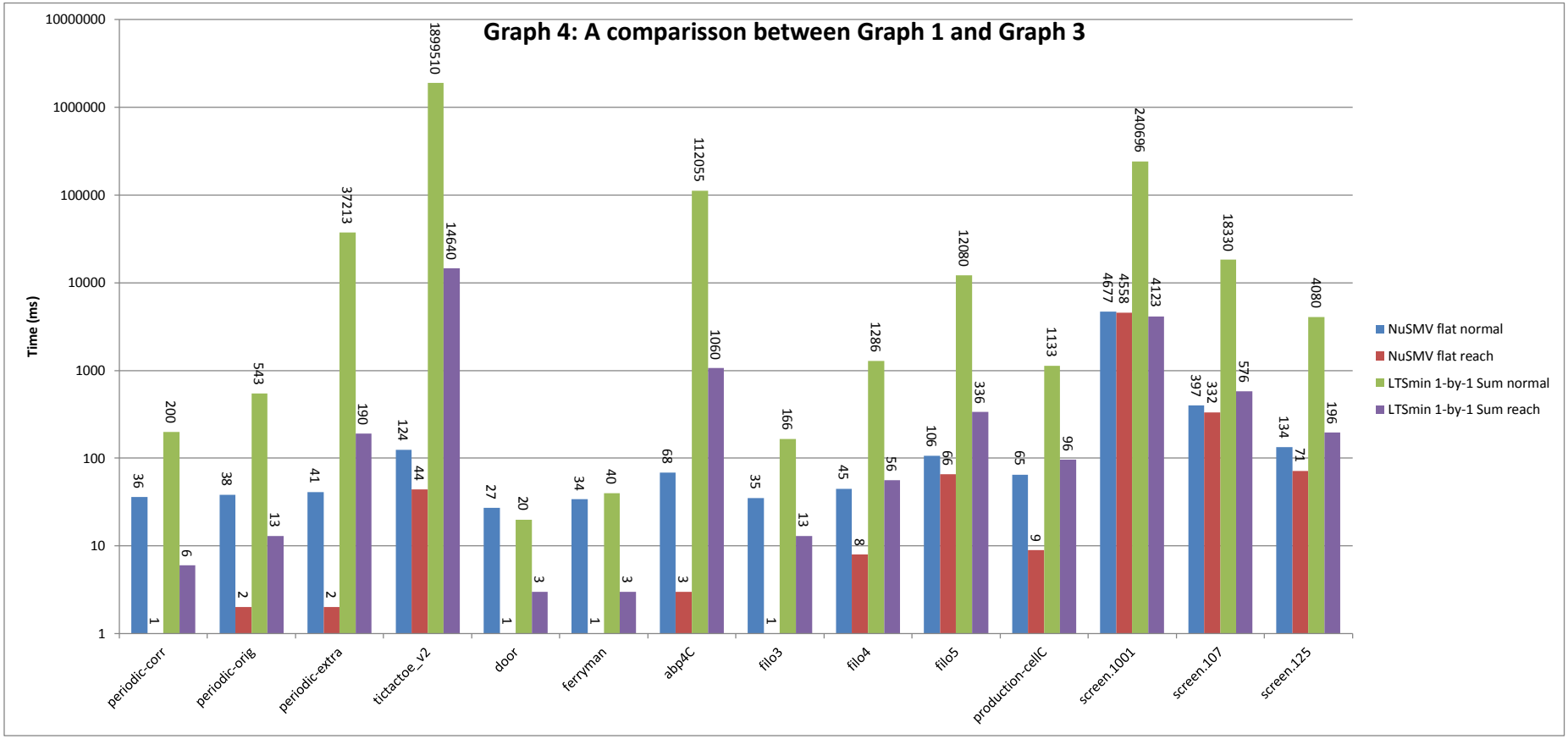
The measurements include the time it takes to parse the SMV file with our created ANTLRv4 SMV parser, the time it takes to translate from SMV to SMV-flat and the time the created software takes to execute the SMV-flat to All-in-1 and SMV-flat to 1-by-1 translations. The x-axis shows the recorded execution times per input model. The y-axis shows the time in milliseconds. SMV-flat to mCRL2 All-in-1 measurements are not available for the abp4C, filo3, filo4 and filo5 models as the translation tool ran out of memory during the creation of the translated models. Please note that the y-axis uses a logarithmic scale base 10.

Graph 2 shows that together with the fact that the data for the SMV-flat to All-in-1 translation is missing for abp4C, filo3, filo 4 and filo5 models because of the mcrl22lps tool running out of memory (after for example 30 minutes of computation time) it is clear that the SMV-flat to All-in-1 translation take an unacceptable amount of time. The explanation of the time usage by this translation is that it generates an **mCRL2 summand** for each possible combination of non-deterministic choices in the model. Each mCRL2 summand is very big as it contains the next state valuation expressions for all variables in the model. The translator software therefore generates big files which take long to create and handle. The 1-by-1 translation was partly designed to counteract the model's size explosion that was seen by the All-in-1 translation. The 1-by-1 translation succeeds as the models are kept manageable in size which is reflected in the translation times. The time taken by the translator software's SMV parser is relatively high, but as the time needed by the SMV parser used by the NuSMV software is embedded in the execution times measured in Graph 1, we have clear proof that the parser's performance can be improved upon. It is therefore not a lasting problem and the current parser performance is good enough for our own experiments.

3.4 Measuring the execution time of the NuSMV and LTSmin reachability algorithm

One explanation for the differences in performance between the LTSmin and NuSMV symbolic reachability tools is that they both solve a very different problem: NuSMV has an input file from which the transition BDD can be easily build. LTSmin builds the transition BDD by way of querying the language module while doing the reachability search (by way of the PINS interface). In order to see how this difference in LTSmin influences its performance a version of LTSmin's reachability function was created that first executes normal reachability and then executes reachability again with the already learned transition BDD. NuSMV was also recompiled with a timer on its reachability function. Graph 3 (on the next page) shows the reachability time of the NuSMV and LTSmin reachability algorithms for the original SMV model and all obtained models through automatic translations. Graph 4 shows a comparison between Graph 1 and Graph 2: The x-axis shows for both the SMV-flat translated model and the All-in-1 translated model the reachability tool runtime compared with the actual reachability algorithm's runtime per input model. The y-axis shows the time in milliseconds. Graph 3 shows the best results found working with option sets that do not include saturation for LTSmin. Reason was that measuring LTSmin while enabling saturation incurred a large measuring overhead and therefore rendered results that where not representable to reality. Graph 3 therefore only gives an indication of the differences in reachability algorithm time and not an exact one when comparing Graph 3 with Graph 1. For Graph 4 we used the no-cluster option package for all LTSmin related measurements shown. Please note that the y-axis uses a logarithmic scale base 10. All zero entries have been altered to 1 ms due to technical difficulties. (This applies to all values that have a 1 ms valuation).


Graph 3: NuSMV VS LTSmin reachability algorithm time
Time (ms)
100000
10000
1000
100
10
1
periodic-corr
periodic-orig
periodic-extra
tictactoe_v2
door
ferryman
abp4C
filo3
filo4
filo5
production-cellC
screen.1001
screen.107
screen.125
NuSMV input
NuSMV flat
LTSmin All-in-1
LTSmin SUM
LTSmin COPY


Graph 4: A comparisson between Graph 1 and Graph 3
Time (ms)
10000000
1000000
100000
10000
1000
100
10
1
periodic-corr
periodic-orig
periodic-extra
tictactoe_v2
door
ferryman
abp4C
filo3
filo4
filo5
production-cellC
screen.1001
screen.107
screen.125
36 1 200 6
38 2 543 13
41 2 37213 190
124 44 1899510 14640
27 1 20 3
34 1 40 3
68 3 112055 1060
35 1 166 13
45 8 1286 56
106 66 12080 336
65 9 1133 96
4677 4558 240696 4123
397 332 18330 576
134 71 4080 196
NuSMV flat normal
NuSMV flat reach
LTSmin 1-by-1 Sum normal
LTSmin 1-by-1 Sum reach

Graph 3 shows that the NuSMV reachability algorithm is significantly faster than the LTSmin reachability algorithm for all examples except the Sokoban puzzles. The data for the Sokoban puzzles (screen 1001, 107 and 125) shows that NuSMV and the 1-by-1 translated models are evaluated using the same amount of time. The All-in-1 translated models of the Sokoban puzzles are evaluated significantly faster by LTSmin then their NuSMV counterpart. The results in Graph 3 one again confirm by way of the periodic examples that NuSMV is not influenced by the extra non-deterministic choices whereas the translated mCRL2 models are. The Sokoban puzzles also support that claim as they all only have 1 non-deterministic choice with 4 options. As such, the Sokoban models must present a difficulty for the NuSMV toolset other than a high (or low) degree of non-determinism.

Graph 4 shows the comparison between a model's evaluation time when the transition BDD relation must first be learned against a model's evaluation time if the transition BDD is already known. The data in Graph 4 shows that LTSmin spends most of it's time creating the transition BDD via the PINS interface. The results in Graph 4 show us that for the 1-by-1 Sum translated models on average, only 4.5% of the time spend in the LTSmin reachability function is used on actual reachability (the rest on the learning of the transition BDD). Graph 4 also shows that NuSMV does not spend all its time in its reachability function. This can possibly be explained by the fact that this is only the case for the small examples (which have a lot of program overhead). The larger examples such as the Sokoban puzzles show that almost all time is spend in the NuSMV reachability function. Section 3.5.2 provides profiling results in order to further clarify how the program spends its time in the tictactoe_v2 and screen.1001 examples.

Comparing the NuSMV results for the models tictactoe_v2 and screen.1001 we see that the first model has 887k states and the second 510k states which is strange as NuSMV spends about 100 times more on the screen.1001 model than on the tictactoe_v2 model. The LTSmin results show that for LTSmin it works the other way around: For the 1-by-1 Sum translation LTSmin finds the screen.1001 model significantly easier than the tictactoe_v2 model. The characteristics found in Table 3.1 and Table 3.2 show that the Tictactoe_v2 model has 3x3x3x3 = 6561 non-deterministic choices whereas the screen.1001 model only has 4 non-deterministic choices. Looking at other models such as the periodic models it does not seems as NuSMV is clearly influenced by the number of non-determisitic choices. LTSmin is influenced however as the translations to mCRL2 translate non-determinism directly into complexity and size of the models. As such, we see for LTSmin that models with high non-determinism often have a lot of dependency groups (as a lot of summands are created). We assume that a trade-off exists in which a larger number of groups exceeds the benefits that LTSmin can create by having those dependency groups. In relation to Graph 3 our experiments show that for the All-in-1 translation LTSmin has to deal with only 8 groups for the screen.1001 model as opposed to 163 groups for the tictactoe_v2 model. For the 1-by-1 Sum translation the screen.1001 model has more groups then the tictactoe_v2 model: 55 versus 33, possibly explaining why the difference in time for the 1-by-1 Sum models is much less significant than for the All-in-1 models.

It is also very likely that the tictactoe_v2 simply has a worse dependency matrix than the screen.1001 model. This would only be affecting the 1-by-1 Sum models as the All-in-one models have a very bad dependency matrix anyway. Tictactoe_v2 may have a worse dependency matrix than the screen.1001 model because the Sokoban models have good locality: All variables represent grid points in the puzzle which only depend on their neighbours or their neighbour's neighbours. The tictactoe_v2 model has worse locality as there are two variables that depend on all variables: There is a variable which determines the game status and a variable which determines the winner which both use all other state variables to determine their next state value. The results for the 1-by-1 Sum Copy translation tell us a different story. All dependency

matrices created with the 1-by-1 Sum Copy translation should be better than those created by the 1-by-1 Sum translation. Seeing as the 1-by-1 Sum and the 1-by-1 Sum Copy models give us opposite results it is clear that further investigation is needed (future work).

Let's now try to explain the difference in results for NuSMV. NuSMV generally finds the Sokoban puzzles very difficult to analyse. A possible explanation is that NuSMV is unable to find a good early quantification ordering (Section 1.2.2) for the Sokoban puzzles compared to all other models. As explained in the section about conjunctive partitioning it is important for the early quantification system to find an order for the state variables in which to remove them from the process of calculating all next states. The SMV descriptions of the Sokoban puzzles contain however a state variable for each grid point in the puzzle which all depend on each other. Often, variables representing neighbouring grid points depend on each other in a circular manner. It is therefore very likely that NuSMV is unable to find a good ordering for the early quantification mechanism effectively bringing down efficiency.

3.5 Other comparison methods

In order to further investigate and possibly clarify the comparison results found through the use of the automatic translations a number of different comparison methods where also executed. This section reports on those comparison methods and the results obtained through them.

3.5.1 Profiling LTSmin

Profiling is a way to see how a program spends it time. An interesting question for example is how much of the time is spend in the language module and how much in the actual reachability. The results of profiling is a list of all functions in the program and the percentage of time spend in them. For the LTSmin reachability tool called "lps2lts-sym" we investigated the periodic-extra, abp4C and screen.1001 models with profiling. For all models we used the 1-by-1 sum version of the translated mCRL2 model. Table 3.5 shows the top five functions in which lps2lts-sym spends its time for the profiled models. Table 3.6 shows the percentage of time spend on different categories. Each function's category is determined mainly by its name-space or by personal experience. Only the first 100 functions mentioned in the profiling report are used to obtain the numbers in Table 3.6.

Table 3.5: LTSmin profiling results by top-5 most used functions

	mdd	mdd		mdd	mdd	mdd	ltsmin
	create_node	sweep_bucket	hash	put	collect	project	find_mcrl2 index
periodic-extra	26.15	11.74	8.39	8.35	2.61	0.06	1.7
screen.1001	9.86	4.97	9.5	2.03	2.29	9.45	3.5
abp4C	17.78	4.83	7.12	4.84	0.9	0.07	3.77

Table 3.5 shows that among the 6 most used functions 5 of them start with "mdd" meaning that they are part of the BDD package. This is no surprise as both reachability and the construction of the transition BDD use BDD operations. Table 3.6 shows that all profiled executions heavily leaned on the BDD package

Table 3.6: LTSmin profiling results by category

	mdd	mcrl2	aterm	boost
periodic-extra	51.02	3.54	3.52	3.42
screen.1001	40.53	4.64	5.75	7.59
abp4C	30.08	5.95	6.2	7.85

(30 to 51% of time spend). It also shows that the axillary libraries of the mCRL2 language module, an aterm library and a boost library al take their fair share in the execution time. They all take up around 5% of total time spend. Note that the sum of the categories does not end up to 100%, this is because not all functions where identified as part of a category and only the 100 most used functions where taken into account. One particularly interesting result is that the execution of the screen.1001 model relies more strongly on the mdd_project function than the others. This is interesting as the screen.1001 model (and also the other Sokoban puzzles) are the functions where LTSmin can compete or even be faster then NuSMV when only looking at reachability algorithm execution time as depicted in Graph 3.

3.5.2 Profiling NuSMV

Profiling NuSMV might tell us more about how the NuSMV program spends its time. For the Sokoban puzzles Graph 4 shows that NuSMV spends all its time in the reachability function. For tictactoe_v2 however the reachability function only uses about one third of the available time. Table 3.7 shows the profiling results for the screen.1001 and tictactoe_v2-flat models. Note that the BDD package of the NuSMV tool is called cudd. Accurately profiling NuSMV is harder than profiling LTSmin as our profiling tool works by measuring intervals of 0.01 seconds. For most NuSMV runtimes we have encountered this interval was too big, making the results more unreliable. The screen.1001 example does not suffer from this problem.

Table 3.7: NuSMV profiling results

	cudd UniqueIter	cudd Cache Lookup	cudd BddAnd AbstractRecur	cudd Cache Lookup2	cudd BddAnd Recur	cudd Other	Other
screen.1001	32.01	13.52	13.12	10.93	10.93	19.21	0.28
tictactoe_v2-flat	4.76	9.52	14.29	4.76	4.76	47.63	14.28

The profiling results for screen.1001 show that all time is spend in BDD operations (cudd). These are exactly the results as suggested by Graph 4, in which it becomes clear that 98.6% of the time spend by the NuSMV executable is used for actual reachability. Those same results suggest that only 32.2% of the NuSMV execution time is used for reachability when it comes to the tictactoe_v2 model. The profiling possibly suggest something different: they suggest that 85.7% of the time is used by the cudd library, which might also be for building the transition BDD. We should remember however that the profiling of the tictactoe_v2 example is fairly unreliable as its runtime is too short.

Chapter 4

Conclusions and future work

This Section summarizes the conclusions that can be drawn from the results presented in Chapter 3. A number of conclusions can be drawn, which of course all depend on the translations described in this work. Other translations and or translations that translate from LTSmin supported input languages to the SMV language might show different results. This section also elaborates on possible future work.

4.1 Conclusions

The NuSMV reachability tool is significantly faster than the LTSmin reachability tool. Graph 1 in Chapter 3 shows that in all cases that have a significant runtime NuSMV is orders of magnitude faster than LTSmin. The difference would only get worse if we would have included the preprocessing steps needed to translate the mCRL2 models into an LPS model. We also did not include an additional mCRL2 related compilation step in the LTSmin reachability tool. The results have been checked by creating an SMV to PROMELA (another LTSmin supported language). The measurements done on the SMV to PROMELA translation confirmed the results shown in Graph 1.

There is no clear winner between the NuSMV reachability algorithm and the LTSmin reachability algorithm. Graph 3 in Chapter 3 shows that for some models the NuSMV reachability algorithm performs better whereas it also shows that for other models the LTSmin reachability algorithm performs better. The difference between the reachability algorithms is that NuSMV uses a conjunctively partitioned transition BDD whereas LTSmin uses a disjunctively partitioned transition BDD. Our results do not show a winner between those two ways of partitioning the transition relation and their related reachability algorithms.

The difference in performance between the NuSMV reachability tool and the LTSmin reachability tool can be explained with the time it takes for LTSmin to learn the transition relation. The data behind Graph 4 in Chapter 3 shows that on average only 9.2% of the LTSmin execution time is spend on actual reachability. As we only measure both the learning of the transition relation and the actual reachability we must conclude that LTSmin spends on average 90.8% of it's execution time learning the transition relation. These results where also confirmed with the SMV to PROMELA translation. NuSMV does not seem to have this problem, at least not for the Sokoban puzzles. The tictactoe_v2 and abp4c models do however seem to have an unexplained difference between its total execution time and the

reachability algorithm execution time. We should however mention that in the total execution time we also include the time it takes for NuSMV to parse the SMV file, something that we do not measure for the LTSmin tool. In combination with the fact that the runtimes of the tictactoe_v2 and abp4C models are still very short (100 - 350 ms) we cannot draw any conclusions for NuSMV.

There is no difference in the results when using either asynchronous or synchronous input models. All results in Chapter 3 show that there is no difference to whether we use an asynchronous or synchronous input model. The asynchronous models used in our experiments where the abp4, filo3, filo4 and filo5 models. Although their results are not comparable to the synchronous Sokoban puzzle model's results (screen. models), they are comparable to the remaining models which are also synchronous models. We can explain this indifference as the asynchronous models are internally by NuSMV and by our tool translated to synchronous models by adding a non-deterministic choice and a number of conditions (the SMV to SMV-flat translation). We do feel that it matters how much non-deterministic choices there are in a model but as the asynchronous models only add one non-deterministic choice to the model there is no real measurable difference as the model may already have numerous non-deterministic choices.

The All-in-1 models are evaluated faster then the 1-by-1 models. The All-in-1 translation is both a success and a curse: On one side the produced All-in-1 model is faster in all cases, on the other side the translation often is not possible for larger models due to time and memory problems occurring in the translation tool and the mcrl22lps tool. The time and memory problems are easily explained with the fact that the All-in-1 translation generates a large mCRL2 summand for each element in the Cartesian product of all non-deterministic choices and their options. This Cartesian product blows up fast as when handling models with multiple non-deterministic choices and their non-deterministic options. Once translated, the All-in-1 model does however produce a non-inflated state space when comparing it to the state space of the SMV-flat model. The 1-by-1 models have on average a state space that is 762 times bigger than the All-in-1 state space. The great difference in the number of states explains why the reachability analysis of the All-in-1 models is faster than the 1-by-1 models.

A linear increase in the number of states does not entail a linear increase in the reachability time Graph 1 in combination with Graph 3 (both in Chapter 3) show that even though the state space size of an 1-by-1 model may be 221 times that of the All-in-1 model(for the periodic-extra example), it does not take 221 times longer to evaluate the state space of the 1-by-1 model. In fact, the time needed to evaluation the 1-by-1 model only increases 1.16 times for the periodic-extra input model. On average (for those models that we could translate with the All-in-one translation) we see a state space increase of 141.4 and a time increase of 79.6. The non-linear relation can be explained with the fact that we are using BDD structures for the transition relation and the visited set. As operations on the visited BDD for example can happen on a compressed groups of visited states at a time it does not necessarily means that a larger group (more states) means a linear growth in the run time for an operation.

4.2 Future work

Collect results from more input models and input models that have a larger state space. A further investigation of the relation between model characteristics and its performance in NuSMV and LTSmin could facilitate a comparison between conjunctive and disjunctive partitioning of the transition relation. Previously determined interesting model characteristics are the degree of non-determinism and locality of the model. An other interesting model characteristic is the average out-degree of the states in the model. The out-degree of a state is the amount of transitions originating from the state. A detailed analysis of NuSMV's early quantification performance and LTSmin's dependency matrix for different models could also provide more insight in which models work good with conjunctive partitioning and which models work good with disjunctive partitioning. Investigating more models that have a significant NuSMV runtime would also facilitate better profiling of NuSMV as to possibly explain the total program runtime versus the reachability algorithm runtime for the tictactoe_v2 and abp4C models.

A further investigation of LTSmin's transition relation learning process. We have concluded that the LTSmin reachability tool is significantly slower than the NuSMV reachability tool. We also found out that all the time difference is used by LTSmin for the discovery and creation of the transition relation. Future work could therefore entail the further analysis of the transition relation learning process in the hope to find unnecessary inefficiencies or improvements in both the theory and implementation.

Compare NuSMV and LTSmin by using translations from LTSmin supported input languages to SMV. In this work we only translated from SMV to LTSmin supported input languages. As such it is possible that we have given an unfair advantage to NuSMV as we only used models that where designed to be used with NuSMV. Future work therefore could include the usage of the S2N tool which translations from a PROMELA subset to SMV to see if it influences the comparison. Automatic translations from both mCRL2 and PROMELA to SMV are possible. It is also possible to ask experts to create a model both in an LTSmin supported language and in SMV in order to compare performance.

Translate SMV specifications to mCRL2 specifications. The SMV-flat to All-in-1 and SMV-flat to 1-by-1 translations have been developed knowing that we only wanted to compare the NuSMV and LTSmin tools for their reachability capabilities. The translation of specifications from SMV to the translated mCRL2 models would greatly enhance the practical use of the created translations.

Create other translations that translate from SMV to LTSmin supported languages. It is possible that the SMV-flat to All-in-1 and SMV-flat to 1-by-1 translations are not the best possible translations. A translation that directly translates from an asynchronous SMV-flat model to an mCRL2 model while encoding the scheduler choices directly into mCRL2 summands instead of an SMV scheduler variable comes to mind. Future work could also focus on translating to other LTSmin supported languages such as the DVE language (from the DIVINE model checker).

(Translation Software:) Generate directly to LPS format instead of mCRL2 format. The translator tool currently generates mCRL2 models which then have to be translated to LPS format by the mcrl22lps tool from the mCRL2 toolset. For the All-in-1 models the tool takes a very long time as the models may get very large. The mcrl22lps tool could be made obsolete if the translation tool would generate the models directly in LPS format.

Chapter 5

The translation software

The translation software used to obtain the results presented in this Thesis was developed specifically for this project. The software is made for scientific purposes only and does not have a commercial quality, meaning that any new models tried with are likely to result in the appearance of bugs. The software was written in JAVA and comes in a JAR package. The software has the following features:

- A custom ANTLRv4 SMV parser.
- An implementation of the SMV to SMV-flat translation.
- An implementation of the All-in-1 and 1-by-1 SMV-flat to mCRL2 translations. For the 1-by-1 translation not only the classic version, but also the SUM and the SUM COPY versions are available, for both translations an option is provided to use either the "substitution" or "functional" approach to translating SMV define macro's.
- An experimental implementation of an SMV-flat to PROMELA translation.
- Mechanics enabling the execution of NuSMV, mCRL2 and LTSmin tools on both Windows and Linux.
- An experiment manager that automatically generates and executes tests for the available translations and the available options (options described in Chapter 3).
- A results collector that obtains and groups together measurements (up to 72 measurements per model).
- A build in Sokoban puzzle to SMV translator.
- A graphical user interface.

The provided graphical user interface ties all features together by providing:

- A file (input model) selection screen.
- A selection screen with options for all available translations and options. Any chosen work will be added to the system as a "job".
- A wait queue that displays jobs that are waiting to be executed.
- A results queue that displays executed jobs.
- Multiple result panels that upon job selection from either queues display a number of intermediate models and or measurement results.

The software is able to run tests using NuSMV, mCRL2 and LTSmin executables. In order to provide compatibility with other computers all needed paths to external executables are encoded in a configuration file called “config.conf”. The paths should point to the actual executable and not to the folder that they are installed in. If the configuration file is missing upon start-up of the software then it is automatically created after which the user is expected to fill in any needed paths.

Bibliography

[1] S.B. Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, 100(6):509–516, 1978.

[2] M. Baldamus and J. Schröder-Babo. p2b: A translation utility for linking Promela and symbolic model checking (tool paper). *Model Checking Software*, pages 183–191, 2001.

[3] S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. *Correct Hardware Design and Verification Methods*, pages 35–50, 2003.

[4] S. Blom, J. Pol, and M. Weber. Bridging the gap between enumerative and symbolic model checkers. 2009.

[5] S. Blom and J. van de Pol. Symbolic reachability for process algebras with recursive data types. *Theoretical Aspects of Computing-ICTAC 2008*, pages 81–95, 2008.

[6] S. Blom, J. van de Pol, and M. Weber. Ltsmin: Distributed and symbolic reachability. In *Computer Aided Verification*, pages 354–359. Springer, 2010.

[7] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[8] J. Burch, E.M. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. *Computer Science Department*, page 435, 1991.

[9] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *Proceedings of the 34th annual Design Automation Conference*, pages 728–733. ACM, 1997.

[10] R. Cavada, A. Cimatti, C.A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. Nusmv 2.4 user manual. *ITCIRST*, 1998.

[11] JG Cleary. Compact hash tables using bidirectional linear probing. *Computers, IEEE Transactions on*, 100(9):828–834, 1984.

[12] P. Dillinger and P. Manolios. Bloom filters in probabilistic verification. In *Formal Methods in Computer-Aided Design*, pages 367–381. Springer, 2004.

[13] R. Hojati, S.C. Krishnan, and R.K. Brayton. Early quantification and partitioned transition relations. In *Computer Design: VLSI in Computers and Processors, 1996. ICCD'96. Proceedings., 1996 IEEE International Conference on*, pages 12–19. IEEE, 1996.

[14] Alfons Laarman, Jaco Van De Pol, and Michael Weber. Parallel recursive state compression for free. In *Model Checking Software*, pages 38–56. Springer, 2011.

[15] Luca Mearelli. Integrating ASMs into the software development life cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997.

[16] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 154–158. IEEE Computer Society, 1995.

[17] H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 130–133. IEEE, 1990.

[18] J. Yong and Q. Zongyan. S2n: Model transformation from SPIN to NuSMV (tool paper).