Acta Cybernetica 15 (2002) 417–455.

Definition of a Parallel Execution Model with Abstract State Machines*

Zsolt Németh†

Abstract

Languages, architectures and execution models are strongly related. A new architectural platform makes necessary to modify the execution model in order to exploit all the advantages of the underlying architecture while preserving its main characteristics. The latter issue requires a careful analysis of the design process. Abstract State Machines offer a powerful method for aiding complex system design. In this paper some aspects of its application are presented by taking the redesign process of a parallel Prolog model as an example.

1 Introduction

The research work presented in this paper aimed at the design of a Prolog interpreter on a multithreaded architecture. However, the certain project represents just the framework and the goal is more general: investigating how a dataflow based model fits a kind of hybrid multithreaded architecture and what the conditions of efficient work are. In a wider scope it deals with the relationship of computational models and the underlying physical architecture.

LOGFLOW is a fine-grained all-solution parallel (reduced) Prolog system for distributed memory architectures. Its abstract execution model called Logicflow [13] can be considered as a sort of macro dataflow scheme, whereas its abstract machine model is the Distributed Data Driven Prolog Abstract Machine [14] (3DPAM). 3DPAM tries to make a connection between a dataflow based execution model and a kind of von Neumann physical architecture.

A hybrid multithreaded platform offers the possibility of creating a more efficient Prolog abstract machine. Its ability to hide latencies due to remote memory access or synchronisation (multithreading) opens a new way for representing Prolog data (heap) and managing the variables. On the other hand, its hybrid feature, i.e. support for both the fast sequential and dataflow execution, is close to the

*This research was supported by the National Research Grant (OTKA) registered under No. T-022106. Presented at the Conference of PhD Students in Computer Science, July 20-23, 2000, Szeged.

†MTA SZTAKI Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, P.O.Box 63. H-1518, E-mail: zsnemeth@sztaki.hu

macro dataflow model of LOGFLOW and makes possible an efficient realisation of dataflow nodes and token flows. To exploit the latter property at the abstract machine level, a new abstract execution model is necessary, too. The new execution model has been derived from the Logicflow in three major steps by changing the way how solution streams are separated, the way how solutions are propagated and by grouping together elementary nodes [18]. Whereas the gain in efficiency is obvious (qualitatively), it is not the case for correctness and semantical equivalence of the models.

The work presented in the paper is a study on the application of a formal method called Abstract State Machines in proving the correctness of the redesign. Abstract State Machines (Gurevich's ASMs, formerly known as evolving algebras) offer a way for the design and analysis of complex hardware and software systems [3] [9]. They are similar to Turing machines in a sense that they simulate algorithms yet, they are able to describe semantics at arbitrary levels of abstraction. An ASM consists of a finite set of transition rules by which the system is driven form state to state, each represented by sets with relations and functions (algebras). By refinement steps a "more abstract" model can be turned into a "more concrete" one and by relating their states and transition rules (by proof mapping) their relative correctness and completeness can be proven. In several refinement steps the equivalence of the models can be shown.

The refinement technique is applied at deriving the new execution model via a series of submodels. LOGFLOW is modeled as an ASM and modifications are introduced by successive new ASMs where each modification step can be checked. Furthermore, implementation steps, creating an interpreter engine can be conducted and checked in the same way.

In Section 2 the notion of computational models are introduced and the circumstances are explained why the modification of Logicflow became necessary. It also summarises the main steps of redesign. Section 3 is a brief introduction to ASMs and their applications. Section 4 puts the design into the framework of ASMs: the initial model and the first derivation are introduced. Finally, in Section 5 the correctness of the first modification is shown.

2 Computational models

Computational models are considered as a higher level of abstraction above languages and architectures [22]. In the course of LOGFLOW project a highly abstract, dataflow based parallel and distributed model called Logicflow [13] has been derived from Prolog language (Figure 1.a). Target architectures were represented of parallel von Neumann types, primarily transputers and networks of workstations. The abstract execution model cannot be implemented directly on the physical machine model but a virtual machine, the so called abstract machine layer is introduced between the execution model and the physical machine model. This way of execution via abstract interpretation is general in case of Prolog and declarative languages.

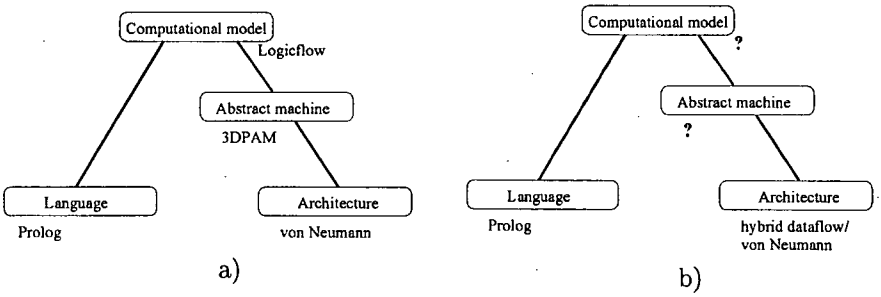

Figure 1: Levels of abstraction

However, the semantical gap between the models to be layers is still too big. The abstract machine, 3DPAM [14] provides the dataflow features required by the execution model at a high cost: token handling, queues, synchronisation, remote communication are realised by software.

Multithreaded architectures offer a solution for fundamental issues of distributed computing: eliminating idling at remote memory access and synchronisation [2]. An emerging class called hybrid dataflow/von Neumann tries to combine the speed of sequential execution and the simplicity and performance of dataflow scheduling [20]. The runtime model of hybrid dataflow/von Neumann architectures is close to that of Logicflow therefore, a natural step is making an attempt to replace the architecture to a hybrid one.

Hence the direction of engineering is reverse with respect to that of the LOGFLOW system: how the abstract engine can exploit the advantages of the architecture. Then how the execution model should be modified in order to fit the abstract engine making a real connection between the language and the architecture (Figure 1.b)?

The multithreaded and the hybrid properties of the architecture are completely independent. Multithreading enables remote memory accesses and thus, allows a new way of Prolog data layout. The main points of the new variable handling and some performance considerations have been presented in [16] and [17]. The hybrid property gives an opportunity for a new and efficient realisation of a Logicflow based model, where all the dataflow features are supported by the architecture. These features can be exploited at abstract machine level but accordingly, the Logicflow model must be be modified, too. The main steps of the modification in the abstract execution model has been presented in [18].

Yet, a set of very important questions remains open: how the original Logicflow and the modified Hybrid (Multithreaded) Logicflow models are related. Are they functionally equivalent? Does the Prolog Abstract Machine exactly what the execution model requires? Is the model sound? In this paper a part of the design is introduced in the framework of ASMs that shows how these issues can be handled and how the design process can be made precise and well documented by a proper formal method.

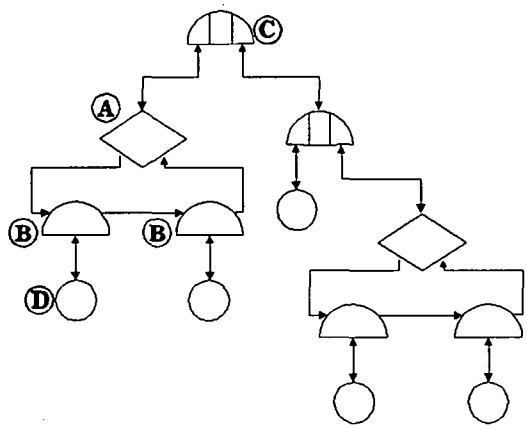

Figure 2: Elements of a DSG graph: Unify (A), And (B), Or (C) and Unit (D) nodes.

2.1 The Logicflow concept

The Logicflow model is a higher abstraction of dataflow principles [13] for a massively parallel (Or and pipeline-And) all-solution execution of Prolog programs on distributed memory architectures. Prolog programs are transformed into a Dataflow Search Graph (DSG, Figure 2). Nodes in this graph represent specific Prolog activities like unification, facts, handling alternatives, etc. Essentially they group together elementary dataflow nodes. As a consequence, DSG nodes can have inner state and one token is always enough to make a node fire.

In this model a clause is represented by a so called Unify-And ring. The Unify node (A in Figure 2) represents the head and the unification, And nodes (B) stand for the body goals and prepare the call. Alternative clauses are connected by Or nodes (C). The example graph in Figure 2 consists of 3 alternatives. Finally, group of consecutive facts are depicted by Unit nodes (D).

Logicflow is a Prolog model without backtrack. Request tokens (representing a query) are propagated from top to bottom. Or nodes duplicate the request tokens and thus, alternative branches of a predicate can be activated simultaneously. In this way Or-parallelism can be exploited. When request tokens reach the Unit nodes, they generate all the possible solutions to the request. Solution tokens form a stream flowing in the Unify-And ring. This ring can be considered as a pipeline: its different stages can process different tokens in the same stream in turn and thus, pipeline And-parallelism can be exploited as well. Solutions are propagated from bottom to top. Nodes must separate different token streams and manage their flow. Due to the all-solution property, there can be hundreds or thousands different token streams, each consisting of several tokens.

2.2 The hybrid dataflow/von Neumann Logicflow concept

The target architecture of the current Multithreaded Prolog Abstract Machine (MPAM) implementation is the Kyushu University Multimedia Processor on Datarol (KUMP/D) [24]. KUMP/D is a successor of multithreaded Datarol [1] and Datarol-II [15] machines. It is a hybrid dataflow / von Neumann one, i.e. it can support both program counter based sequential execution and dataflow scheduling. More precisely, the Datarol execution model distinguishes the short term and long term execution. Short term execution means sequential processing whereas long term execution is dataflow based scheduling of the sequential threads. In such a way there are threads that run exclusively until the termination point sequentially. At the end the next thread is scheduled on dataflow principles. In other words it is a kind of macro dataflow model, too.

In this model a program consists of simultaneously existing function instances. A function instance has its own context (frame) and shared code. Note, that the function instance and the thread are not the same: a function instance may consist of multiple threads. They belong to the same context. According to the definition of the thread, in a single context they do not work concurrently, rather the function can be considered as a set of consecutive threads. A thread is terminated whenever a synchronisation or remote memory access causes latency. At this point a fast context switch allows the processor to go on eliminating idle cycles; it is the essence of multithreading.

In the MPAM model each DSG node could be represented by a frame, where its own context is stored, furthermore it also has registers for token information. There is a thread (or more threads) attached to the frame. In such a way a node is represented as a function instance: the function code is realised by the threads whereas arguments and local variables of the function are kept in the frame.

A running function instance can activate (call) another function. It can pass arguments just like in case of procedure call of other programming models. When the new instance is ready to run, the scheduler may select it for execution. However, at this point, the caller instance remains as it is, its content is not stored in a token (in contrast with 3DPAM model). The new instance can proceed without load operations, because all the necessary data are available as arguments. A passive instance is waiting for some results. When the specified results are ready, it can be awaken on dataflow principles, where no load instructions are necessary: the state is the same as it was before, the results are local variables.

These are the principles of the MPAM working model that ensure an efficient interpretation of Prolog programs on the new architecture. However, the abstract execution model must be modified, too in order to narrow the semantical gap. There are three significant steps of redesign that enable the changes in the abstract machine model: creating node instances, optimised paths at alternatives and the introduction of aggregate nodes [18].

In Logicflow token streams form a central concept. Streams are maintained (and different streams are separated) by a colouring scheme and at abstract machine level the context tables that are needed for keeping consistent the colouring represent

some restrictions. However, the separation of streams could be defined in another way at abstract execution level. Obviously, if it could be guaranteed, that a node emits only one request token, there are no multiple reply streams and thus, they need not be separated. The key is in the Unify-And ring where And nodes prepare calls to predicates, i.e. they emit tokens towards Or, Unify or Unit nodes. If for each token in the stream a new instance of And node is created, the called node beneath it will receive a single request token. The Unify node merges the answer streams from the last And nodes within the ring. They belong, however, to the same stream representing the answer to the single request token of the Unify node. In such a way the token streams are separated physically without the need of colouring.

Or nodes (handling alternatives) do nothing at merging solution streams but maintain the correct colours. If the token colouring scheme can be eliminated, according to previous principles, there is no need to propagate solutions through the cascade of Or nodes, they can reach their root in one step.

As it was set forth the goal is to create an execution model, where nodes can be mapped to function instances easily. Although, it is possible at the present stage, increased granularity would reduce the cost related to instance (frame) management and data transfer between frames. The granularity can be increased by grouping together DSG nodes, resulting aggregate nodes. By a formal analysis 8 types of aggregate node have been defined as: unit, unify, or-unit, or-unify, and-or-unit, and- or-unify, and-unit, and-unify (Figure 3). In an aggregate node the component nodes share context and register information in a single frame saving significant time associated with frame set-up, communication, argument passing, intranode dataflow and so on. Thus, the unit of execution is an aggregate node that can be handled as a function instance with all the optimisations introduced before.

3 System design with ASMs

The idea of transforming the original Logicflow model has been presented in [18]. Yet, it is a rather informal description of the principles. The scope of current investigation is the verification of those transformation steps, in a broader sense, the question should be answered if the Logicflow and the Hybrid Multithreaded (HM) models of Prolog execution are semantically the same. Next, MPAM must be defined in such a way that it is equivalent to the HM Logicflow model. Abstract State Machines are proven to be capable, powerful and especially useful for solving this problem. They are able to deal with the very high level of abstraction of execution models and at the same time they are flexible enough to deal with MPAM at a significantly more concrete (with respect to implementation) level.

3.1 Abstract State Machines

Abstract State Machines represent a mathematically well founded framework for system design and analysis [3][6] introduced by Gurevich as evolving algebras [9].

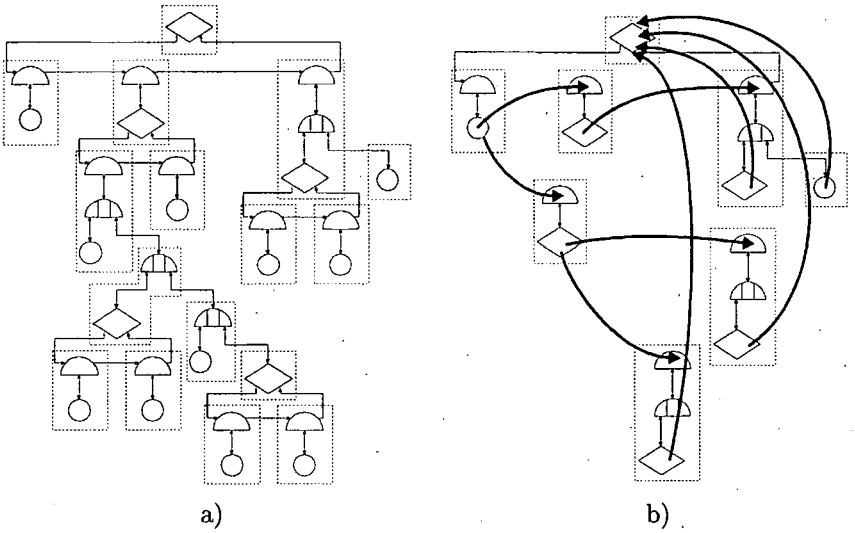

Figure 3: An example DSG graph where aggregate nodes are depicted as grey rectangles. The graph consists of 16 nodes instead of 30. All 8 types of aggregate nodes can be seen here (a). A snapshot of a possible execution showing node instances and optimised return paths (b).

The motivation for defining such a method is quite similar to that of Turing machines (TM). However, while TMs are aimed at formalising the notion of computable functions, ASMs are for the notion of (sequential) algorithms [12]. Furthermore, TMs can be considered as a fixed, extremely low level of abstraction essentially working on bits, whereas ASMs exhibit a great flexibility in supporting any degree of abstraction.

In every state based systems the computational procedure is realised by transitions among states. In contrast with other systems, an ASM state is not a single entity or a set of values but ASMs states are represented as (modified) logician's structures, i.e. basic sets (universes) with functions and relations interpreted on them. Experience showed that any kind of static mathematic reality can be represented as a first-order structure [12]. These structures are modified in ASM so that dynamics is added to them in a sense that they can be transformed.

Applying a step of ASM M to state (structure) A will produce another state A' on the same set of function names. If the function names and arities are fixed, the only way of transforming a structure is changing the value of some functions for some arguments. The transformation can depend on some condition. Therefore, the most general structure transformation (ASM rule) is a guarded destructive assignment to functions at given arguments [3].

ASMs are especially good at three levels of system design [3]. First, they help elaborating a ground model at an arbitrary level of abstraction that sufficiently rigorous yet easy to understand, defines the system features semantically and inde-

pendent of further design or implementation decisions. Then the ground model can be refined towards implementation, possibly through several intermediate models in a controlled way. Third, they help to separate system components. ASM is not a paper theory but it has been applied in various industrial and scientific projects like verification of Prolog [4] and Occam [5] compilers, Java virtual machine [23], PVM specification [7], ISO Prolog standardisation, validating various security and authentication protocols, VLSI circuits, and many more. The definition of ASMs is written in [8] and [11] and a tutorial can be found in [9]. A brief summary is presented here in order to make the paper self-contained.

A vocabulary (or signature) is a finite set of function names, each of fixed arity furthermore, the symbols *true*, *false*, *undef*, $=$, the usual Boolean operators and the unary function Bool. A state A of vocabulary Υ is a nonempty set X together with interpretations of function names in Υ on X. X is called the superuniverse of A. An r-ary function name is interpreted as a function from X^r to X, a basic function of A. A 0-ary function name is interpreted as an element of X.

In some situations the state can be viewed as a kind of memory. Some applications may require additional space during their run therefore, the *reserve* of a state is the (infinite) source where new elements can be imported inside the state.

A location of A (can be seen like the address of a memory cell) is a pair $l = (f, \mathbf{a})$, where f is a function name of arity r in vocabulary Υ and $\mathbf{a}$ is an r-tuple of elements of X. The element $f(\mathbf{a})$ is the content of location l.

An update is a pair $a = (l, b)$, where l is a location and b is an element of X. Firing a at state A means putting b into the location l while other locations remain intact. The resulting state is the sequel of A. It means that the interpretation of a function f at argument a has been modified resulting in a new state. This is how transition among states can be realised. An update set is simply a set of consistent updates that can be executed simultaneously.

ASMs are defined as a set of rules. The simplest rule is the skip that does not do anything. An update rule $f(a) := b$ is a rule and causes an update $((f, a), b)$, i.e. hence the interpretation of function f on argument a will result b. It must be emphasised that both a and b are evaluated in A.

A conditional rule R of form

```
if c then
   R1
else
   R2
endif
```

is a rule. To fire R the guard c must be examined first and whenever it is true R_1 otherwise, R_2 must be fired. A block of rules is a rule and can be fired simultaneously if they are mutually consistent.

An import rule of form

```
import v
   R
```

`endimport`

is a rule for introducing new elements from the *reserve* and firing rule R. The following construct

`extend` U `by` $v_1, ...v_n$ `with`
R
`endextend`

is a shorthand notation for

`import` $v_1, ...v_n$
$U(v_1) := true$
...
$U(v_n) := true$

R
`endimport`

that is new elements are imported from the *reserve* and they are assigned to universe U and then rule R is fired.

There are further rules introduced for convenience but these are the inevitable ones that will be used thoroughly in this paper. The basic sequential ASM model can be extended in various ways like nondeterministic sequential models with the choice construct, first-order guard expressions, one-agent parallel and multi-agent distributed models. The latter is applied in modeling Logicflow, therefore a very brief introduction follows.

A distributed ASM consists of

- a finite set of single-agent programs Π_n called modules
- a vocabulary Υ, which includes each $Fun(\Pi_n) - \{Self\}$, i.e. it contains all the function names of each module but not the nullary $Self$ function
- a collection of initial states

The nullary $Self$ function allows an agent to identify itself among other agents. It is interpreted differently by different agents (that is why it is not a member of the vocabulary.) An agent a interprets $Self$ as a while an other agent cannot interpret it as a. The $Self$ function cannot be the subject of updates.

A run of a distributed ASM is a partially ordered set M of moves x of a finite number of sequential ASM agents $A(x)$ which

- consists of moves made by various agents during the run. Each move has finitely many predecessors.
- The moves of any single agent are linearly ordered.
- Coherence: each initial segment X of M corresponds to state $\sigma(X)$ which for every maximal element $x \in X$ is obtainable by firing $A(x)$ in $\sigma(X - \{x\})$.

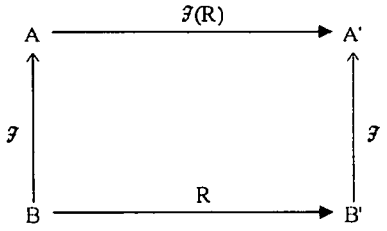

Figure 4: Principle of refinement.

3.2 Model refinement

Refinement is defined as a procedure where a "more abstract" and a "more concrete" ASMs are related according to the hierarchical system design. At higher levels of abstraction implementation details have less importance whereas they become dominant as the level of abstraction is lowered giving rise to practical issues. The goal is to find a controlled transition among design levels that can be expressed by a commuting diagram.

Let us assume ASM **M** has been refined to ASM **M'** by a partial abstraction function $\mathcal{F}$ that maps certain states of **M'** to **M** so that the diagram commutes in Figure 4. To put in another way, if the refinement is correct, it is the same if ASM **M'** moves from B to B' and then the corresponding state of ASM **M** is taken or first $\mathcal{F}(B)$ is taken and then the rule corresponding to R is fired. In both cases the result should be A'. However, the notion of equivalence, correctness and completeness strongly depends on the system designer's needs as it will be shown later.

4 From Logicflow to HM Logicflow model of execution

ASM represents the framework for proving the correctness of the new HM Logicflow model with respect to its predecessor Logicflow. Although the refinement procedure was introduced before as a transition between design levels, it is just a consequence of its "traditional" application. In fact, refinement is a method to relate any two ASMs of any level of abstraction. In our case Logicflow and HM Logicflow models that represent the same design levels, are related.

What has not been mentioned before is the considerable amount of intuition that is necessary at making a refinement step. Making a suitable mapping between corresponding states and rules is a hard task, if not impossible in case of complex systems. Instead, the gap between the two models should be divided by introducing submodels that differ only in one or two properties from the previous one and thus, a simple one-to-one mapping can be applied to some of the rules and states whereas the rest of mapping can be conceived by reasoning.

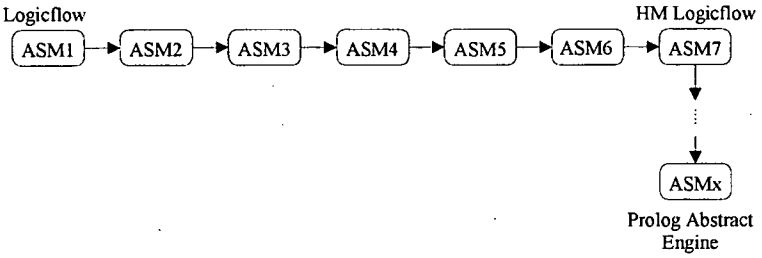

Figure 5: From Logicflow to HM Logicflow by a series of refinement steps.

As it was introduced earlier, the HM Logicflow model can be derived from Logicflow in three transformation steps. Yet, these steps from model to model are still to big because transformations involve the modification or replacement of many features at a time. Finally, a proper set of five submodels was found where the mapping can be done with reasonable efforts (Figure 5).

ASM1 is the original Logicflow model. In ASM2 a new kind of synchronisation is introduced in the Unify-And ring. As a consequence, in ASM3 instances of And nodes can be created. It yields that every node receives just one request token thus, there is no need for token colouring in ASM4. If there are no colours, the cascades of Or node can be optimised in ASM5. In ASM6 the concept of frames are brought into existence whereas ASM7 is the model for the HM Logicflow with all its details [19]. As it can be seen in Figure 5, all these models are at the same level of design abstraction.

The design of MPAM completely fits the same methodology. (It is entirely out of the scope of this paper and can be found in [19].) By successive refinement steps the abstract HM Logicflow model can be turned into a model of the engine that is much closer to the implementation level. The instructions of MPAM can be derived from groups of instructions of the ASM that describes it. It should be emphasised that in this scheme in Figure 5 both the model describing the principle of execution and the implementation details can be designed in the same formal framework of ASMs.

4.1 Description of the Logicflow model by an ASM

Logicflow is a distributed, dataflow and thus, indeterministic execution scheme that can be modeled as a distributed multi-agent ASM. There are two questions to be clarified:

1. From which point of view should the model be described, i.e. what entities should be the agents? The system could be modeled as the graph nodes are agents and react to the incoming tokens. Another possibility, that was chosen finally, is where tokens are agents and they make the nodes fire. Although this issue must be clearly answered before building the model, the decision is rather the question of taste.

2. What should the ASM describe? In [13] the function of each DSG node is described and it is claimed that every Prolog program can be compiled to a set of such nodes. Yet it is not a description of the Prolog execution. For instance the DSG graph for a Prolog program that contains recursion (and most Prolog programs do) may be different for different input parameters although the DSG components (the compiled program) and their functionality are the same. The ASM model aimed at simulating the actual execution, therefore it describes how the certain DSG graph is constructed from the precompiled building blocks.

The machine created for modeling Logicflow is called ASM1. There is just one module thus, each agent executes the same program. Furthermore, the number of agents changes during the execution as tokens are created and discarded. The $Self$ function is realised by the nullary function t, i.e. it means the current token that realises the agent and the same t in the program text is interpreted differently for different agents.

4.1.1 The basic sets and functions

ASM1 consist of the following universes:

- $TOKEN$. Elements in this set are the agents. The nullary function t represents the $Self$ function. Tokens have type and colour. The unary function $type : TOKEN \rightarrow \{DO, SUB, SUCC, FAIL, FAIL2\}$ [1] and $colour : TOKEN \rightarrow COLOUR$ can retrieve the type and colour of the given token, respectively. $loc : TOKEN \rightarrow NODE$ returns the current location (node) of the token. It is assumed that tokens are always assigned to a node and there is no buffering or transition time between two nodes. Some tokens can carry environments, i.e. variable substitutions that can be obtained by the subst: $TOKEN \rightarrow SUBSTITUTION$ function.

- $NODE$. This universe contains the nodes that realise the actual DSG graph. There are 5 types of them that can be retrieved by $node : NODE \rightarrow \{AND, OR, UNIT, UNIFY, QUERY\}$. $mode : NODE \rightarrow \{create, active\}$ is related to the construction of the DSG graph and results if the graph connected to the node has been built already or not. The function $returnport : NODE \rightarrow \{reply.in, reply.in1, reply.in2\}$ gives the port type where the actual node must return the answer tokens. The topology of the nodes can be described by the $on\_arc : NODE \times INT \rightarrow NODE$ function and by the macros derived from it:

  - $child(node) \equiv on\_arc(node, 3)$
  - $child1(node) \equiv on\_arc(node, 3)$

[1] FAIL and FAIL2 tokens are functionally equivalent and they are not distinguished in [13]. The introduction of FAIL2 tokens is simply a notation for making the explanation easier. FAIL2 tokens are those occuring in a Unify-And ring.

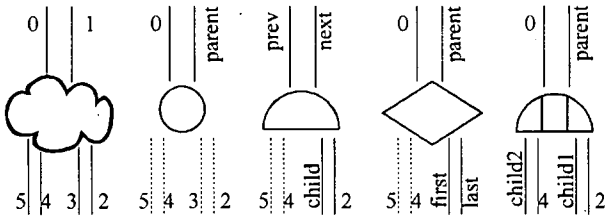

Figure 6: A generic node and interpretations of arc labels for different types of nodes.

- $child2(node) \equiv on\_arc(node, 5)$
- $prev(node) \equiv on\_arc(node, 0)$
- $next(node) \equiv on\_arc(node, 1)$
- $parent(node) \equiv on\_arc(node, 1)$
- $first(node) \equiv on\_arc(node, 3)$
- $last(node) \equiv on\_arc(node, 2)$

This kind of description assumes a generic node with 6 arcs as it can be seen in Figure 6. The actual types of nodes enumerate their arcs accordingly, though not all arcs are in use. In such a way a kind of navigation can be defined among nodes, their relationship (parent-child, previous- next) can be described precisely yet, in a readable form.

Some nodes contain context information like colour, substitution, counter that can be retrieved by the appropriate functions ($colour\_context : NODE \times COLOUR \rightarrow COLOUR$, $subst\_context : NODE \times COLOUR \rightarrow SUBSTITUTION$, $counter : NODE \times COLOUR \rightarrow INT$, $and\_state : NODE \times COLOUR \rightarrow \{open, closed\}$, $or\_state : NODE \times COLOUR \rightarrow \{wait1, wait2\}$).

- *COLOUR*. Token streams are separated by a colouring scheme. Tokens forming a stream have the same colour no matter what the actual type or content of the token is.

- *STREAM*. Tokens of the same colour targeted to the same port of a node form a stream. It is essentially a set. Tokens in this set can fire the node they are waiting for in arbitrary order except that a *FAIL* token must be the last one terminating the stream. A stream can be identified by the node and port the tokens are waiting for and the colour information ($stream : NODE \times PORT \times COLOUR \rightarrow STREAM$). The relation $instream : STREAM \times TOKEN \rightarrow \{true, false\}$ is true if the given token is a member of the stream, whereas function $card : STREAM \rightarrow INT$ returns the number of tokens in the stream.

- $SUBSTITUTION$. Substitution is a set of variables and their binding values.

- $PORT$. A port is an entry point to a node from the following set: $PORT = \{request.in, reply.in, reply.in1, reply.in2\}$.

- $LIT, CLAUSE$. Literals and list of literals, i.e. clauses. Function $procdef : LIT \rightarrow CLAUSE^*$ returns the definition for the given literal. A clause can be separated to head and body parts by the $head : CLAUSE \rightarrow LIT$ and $body : CLAUSE \rightarrow LIT^*$ functions. There is a predicate or goal assigned to some nodes that can be retrieved by $predicate : NODE \rightarrow CLAUSE^*$ and $goal : NODE \rightarrow LIT$, respectively.

4.1.2 Modeling Logicflow by ASM1: an example

A simple example program is presented here step-by step that shows the most important features of the ASM1 model. Abstract State Machines can be treated as a kind of pseudo-code so, even if one is not familiar with all the details of ASMs, the code can be read easily and it is self-explanatory more or less (see Appendix A.)

The example given here is the well-known family program:

```
grandfather(X,Y):-father(X,Z),parent(Z,Y).
parent(A,B):-mother(A,B).
parent(A,B):-father(A,B).
father(bill, john).
father(bill, james).
father(john, jack).
mother(jane, jack).
mother(alice, fred).
mother(jane, charles).
:-grandfather(X, jack).
```

The execution starts with one Do token (agent) at the Query node (Figure 7.a). There are no other tokens or nodes in the system. The activator of the Do token is `grandfather(X, jack)`, i.e. the query, and the substitution is empty. This initial state is represented by the following structure:

$type(loc(t)) = Query$
$type(t) = DO$
$subst(t) = \{\}$
$act(t) = grandfather(X, jack)$

In this case rule 14 can fire extending the graph with a new but untyped node. This operation is realised by the extend construct that brings a new element from the reserve (and this element is different from those already in some basic set) and puts it into a set, $NODE$ in this example. The relationship between the Query

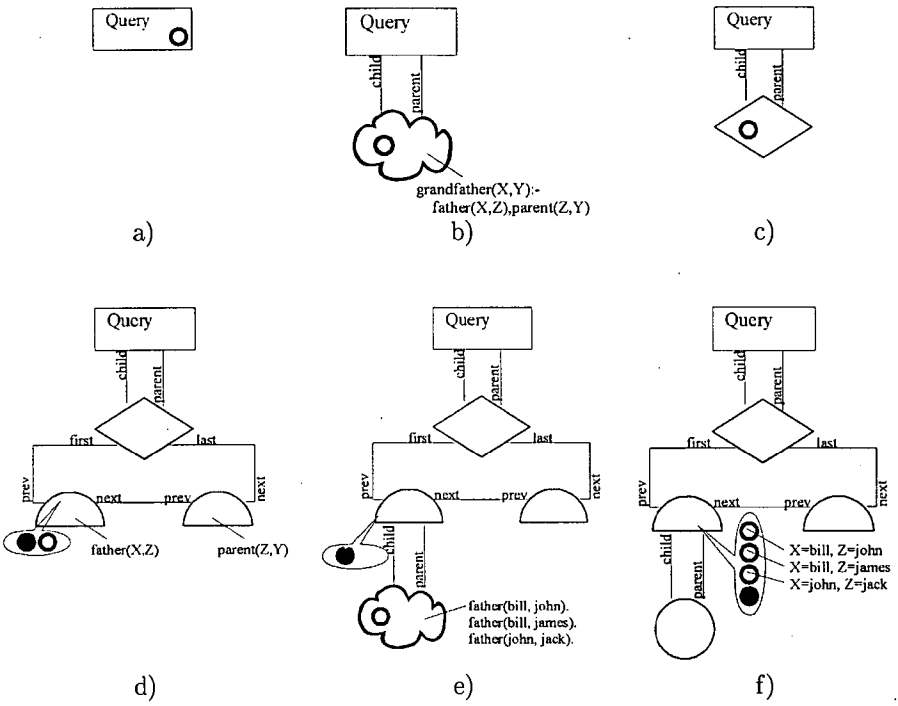

Figure 7: The initial state and states after firing rules 14, 13, 7a, 2a, 13, 3, 1 respectively. The result of rules 13, 3, 1 is shown together in f).

already in play and the new node is set by the macro variations of the *on_arc* function. The predicate assigned to this node is `grandfather(X,Y):-father(X,Z), parent(Z,Y)`and the Do token is moved to it (Figure 7.b). Then rule 13 can fire that sets the type of current undefined node to Unify (the predicate is a single clause having body.) This change enables rule 7a to fire. Note, the Unify node is in *create* mode, i.e. it is the first time a token appears on it and the subgraph must be extended. The Unify node represents the head of a clause where unification takes place. If the unification of the activator of the token is successful with the head of predicate (and it is in this case), the graph is extended by And nodes resulting the Unify-And ring. Each And node in this ring is in *create* mode, their connecting arcs are set and body goals are assigned to them. The current colour and the substitution (updated with θ, the most general unifier) of the token are saved and a new colour is assigned to it. Note that the new colour is obtained by the `extend` construct which guarantees that this colour is different from all previous ones. A stream is created towards the *request.in* port of the first And node in the ring and the current token (transformed into a Sub type, substitution is θ and location is the And node) is put into it together with a terminating Fail2 token (Figure 7.d).

At this point rule 2a can fire. The node sets its counter to 1 (the number

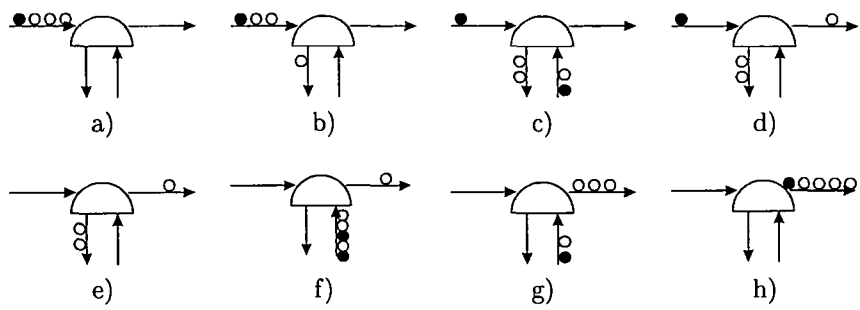

Figure 8: Working cycle of an And node in Logicflow model

of received Sub tokens) and since it is in *create* mode (see the extension in rule 7a) it produces its subgraph. The subgraph is untyped currently, and the assigned predicate is `father(bill, john), father(bill, james), father(john, jack)` (Figure 7.e).

Hence two rules may fire simultaneously. Rule 13 sets the type of untyped node to Unit (a predicate with multiple clauses and none of them have a body), whereas rule 3 sets the state of the And node closed and the Fail2 token vanishes. As a consequence of rule 13, rule 1 can fire. A Unit node brings together the successive facts in the program and produces all the possible solutions to them. It creates a stream to its parent node (i.e. the And node) and puts the solution tokens (Succ tokens) into it. Each Succ token has the same colour and they are identical with exception for the different substitutions according to the result of three different unifications. Finally, a Fail token is put into the stream terminating the computation (Figure 7.f).

The first steps of executing the example program showed the most important features of the ASM1 model that describes the Logicflow model. The reader may trace the execution further by applying the appropriate rules in Appendix A.

4.2 The first submodel: ASM2

The first model that has been introduced between the Logicflow and the Hybrid Multithreaded Logicflow introduces a different way of synchronisation in the Unify-And ring. The basic philosophy of Logicflow is that for a token representing a goal an answer *stream* of the same colour, terminated with a Fail token, is expected at the reply arc of the node no matter if it was produced by a single node or by a large subgraph. The receipt of the Fail token means that all the possible solutions to the query has been found and there are no active tokens belonging to the computation in the subgraph.

Ensuring this property in the Unify-And ring is a complex task. There can be multiple overlapping streams in the ring separated and identified by colours. An And node receives a token stream and must generate a token stream of the same colour making it sure that the Fail2 token appears on its reply arc only when

no more solutions are possible. This is realised by *and_state* and a counter. The counter contains the number of tokens sent to the subgraph, i.e. each incoming request token increments it (Figure 8a, b, c) whereas each terminating Fail token in the answer stream decreases it (Figure 8 d, g, h). The Fail2 token on the request arc of the And node makes its state closed meaning that no more request tokens can be expected (Figure 8 e). At this point whenever the counter is 0, the Fail2 token can be sent on the reply arc terminating the answer stream (Figure 8 h).

This solution can be considered as a distributed tracking of the active streams in the Unify-And ring. A single Fail2 token is circulated in the ring terminating the request/reply stream and whenever it hits the Unify node, there are no more tokens belonging to the same task in the subgraph. (This property has been proven in [13].)

However, if And node *instances* are created for each token in the stream according to the modifications, this mechanism is not viable, since there is no single route for tokens in the ring and thus, there cannot be a single termination signal at the end. The first step in the modifications is the redesign of the synchronisation mechanism in the Unify-And ring.

States and counters in the And nodes, furthermore Fail2 tokens are not necessary anymore. Instead, a counter is introduced in the Unify node that keeps a record of the active streams in the ring. A new type of nodes is introduced as Last_And which is the last and node within the Unify-And ring. Each time an And node receives a solution token, it increments the counter in the Unify node. Each time an And or Last_And node receive a Fail token, they decrement the counter. The functionality of And and Last_And nodes is equivalent except that Last_And nodes never increment the counter.

ASM2 is the model that describes the Logicflow model where this slight modification is introduced. Most rules remain intact except those related to And or Unify nodes. (See Appendix B.)

The modification of the synchronisation mechanism within a Unify-And ring seems to be simple, feasible and correct. But can it be shown formally that ASM1 and ASM2 are equivalent and functionally they do exactly the same?

5 Proof of equivalence of ASM1 and ASM2

This proof represents the first element in the series of equivalence proofs in Figure 5. It is introduced here as a kind of case study and further proofs can be carried out in a similar way. First, it should be clarified what equivalence means. Then it must be defined how the indeterministic behaviour of these models can be treated. While the latter issue is general in the whole proof procedure, the first one is unique for each step, i.e. two model can be said equivalent with respect to some definition. Obviously, these definitions involve the property that has been changed in the given refinement step.

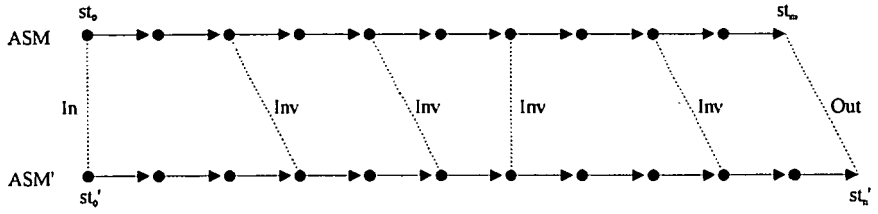

Figure 9: Schellhorn's modularisation theorem

5.1 The notion of equivalence

There can be many definitions of equivalence according to the level of abstraction and it is even possible that two algorithms are identical to some definitions of equivalence and different to others [10]. ASMs offer a possibility to precisely define what equivalence means in the given situations. One presumption for the equivalence is that the two algorithms produce the same output for the same input. In [10] there are two possible equivalencies defined, the strict lock-step equivalence and the lock-step equivalence. It is shown that the two algorithms in scope (variations of bounded buffers) are lock-step equivalent but not strict lock-step equivalent. In both cases every step of algorithms and the corresponding states are taken into consideration which is not feasible for real life complex applications.

A more practical approach that can be applied at refinements is presented in [21]. Let us assume two relations IN and OUT of initial and final states, respectively. A refinement is correct if for every finite trace of $ASM'(st'_0, ...st'_n)$ and for every st_0 of ASM with $IN(st_0, st'_0)$ there exists a finite trace of ASM $(st_0, ...st_m)$ so that $OUT(st_m, st'_n)$. In other words: let us take into consideration all valid runs of ASM' starting from st'_0 and ending in st'_n. For each such run let us take all the st_0 states of ASM that are in relation $IN(st_0, st'_0)$. If every run of ASM starting from st_0 ends in state st_m that is in relation $OUT(st'_n, st_m)$ then the refinement of ASM to ASM' is correct. If the refinement of ASM' to ASM is correct, too, then it is complete. Although, it is just a correctness of a kind of model transformation, it is also a definition of equivalence based on input-output behaviour.

Schellhorn's main invention is the generalised proof method for refinement correctness (the "Modularisation Theorem"). The commuting diagram can be partitioned by finding states that are in arbitrary relation which is the so called coupling invariant (Figure 9). In such a way the correspondence between two computations can be reduced to subcomputations, i.e. if the two ASMs are started from related states they should finish their computation in related states as well. Schellhorn formalised his theorem for deterministic and indeterministic ASMs and defined the trace correctness as well. In the followings Schellhorn's idea is applied for equivalence proof.

5.2 The problem of indeterminacy

Logicflow model (and its modified versions) is inherently indeterministic due to dataflow nature. The corresponding ASM models are distributed multi-agent ones with similar behaviour. A program is deterministic if for some input set it generates the same output set no matter how many times the program is executed. Yet, the execution still can be indeterministic reaching the output set in different ways (different order of state transitions) from run to run. The main problem is that some execution paths can lead to the correct output set while others do not.

Schellhorn's theorem for deterministic ASMs says that if there are two states x and x' that are in relation by the coupling invariant, there exist two integers i and j so that after i step of **ASM** and j step of **ASM'** the coupling invariant holds for the resulted states in order to clain the refinement correct. However, it is just one possible successor state. Shellhorn generalizes his theorem for indeterministic behaviour so that for every possible x'_j (the resulted state after j steps) there must be an i so that x'_j and x_i are in relation.

What does it mean? It is not enough to find one possible partition of the commuting diagram but all the possible partitions. A serialisation method will be used according to [3]: given any initial segment of a run, each linearisation has the same final state. It is a consequence of the coherence condition in the definition of distributed ASMs. The execution is serialised, and then the partitions can be obtained according to the principles of partitioning deterministic systems. In this case no special properties of the actual serialisation can be used, because it is not *one* linearisation but *any* of them. In other words the partitioning will yield all the subdiagrams of which all the possible linearised executions of the two ASMs can be constructed.

5.3 Definition of equivalence of ASM1 and ASM2

Obviously, two Prolog executions are equivalent if they produce the same solutions to a given query. (Due to the all-solution property of the Prolog models in scope and the absence of side-effects the order in which solutions are given is meaningless.) However, taken into consideration two facts, several rules can be omitted at the proof thus significantly reducing the size of the commuting diagram.

First, there are several rules that are identical in ASM1 and ASM2. It is the consequence of careful insertion of submodels where special attention was paid for introducing small changes from model to model. Evidently, they do not affect the equivalence of the two ASMs. Furthermore, ASM rules are local in a sense that they modify the state of the current token and the node it is currently on and do not affect other tokens or nodes in any way.

As a consequence, the equivalence of the two models can be proven by showing the equivalence of the working cycle of Unify-And rings. It can be assumed that the embedding graph behaves the same in the two cases and there are no interactions among different Unify-And rings. First, let us assume that there are no other Unify-And rings in the subgraph attached to the Unify-And ring in question. If they are

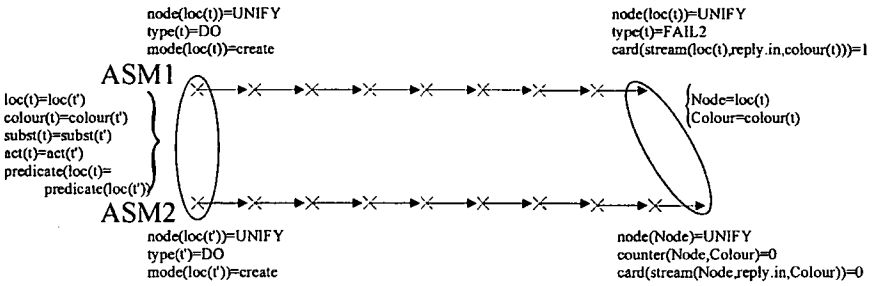

Figure 10: The reduced commuting diagram.

proven to be equivalent, they behave exactly the same as a Unit node with respect to the generation of a token stream and then the equivalence holds in case of any embedding graph.

ASM1 and ASM2 are equivalent if and only if for each valid run of ASM1 there is a related run of ASM2. According to the reasoning above, this definition can be narrowed as: the two models are equivalent if and only if for each valid run of a Unify-And ring in ASM1 there is a corresponding run of the same Unify-And ring in ASM2.

In both cases the initial state is represented by the appearance of a Do token at the Unify node. The final state in ASM1 is the appearance of a Fail2 token on the reply.in arc of the Unify node, whereas the related state in ASM2 is composed by the 0 state of the counter and the empty stream. (Note that in ASM2 there is no Fail2 token on the Unify node, that is why the condition is expressed as a first-order formula in rule 24.) The correspondence is expressed by the same properties of tokens and nodes as it can be seen in Figure 10.

5.4 Partitioning the commuting diagram

In Figure 10 a reduced commuting diagram can be seen. It is reduced in a sense that state transitions occurring in the Unify-And ring are included only. It shows one possible sequence of execution and as it has been explained, the proof should cover all possible execution patterns.

The most important and generally the most difficult task is finding the proper coupling invariant. It can be any property that relates a state of ASM1 to ASM2 and by which the diagram can be partitioned in such a way that from the initial state a related pair of states can be reached in finite steps, then the invariant property holds for some pairs of states, finally, from a related pair of states the final relation can be reached (see Figure 9). The essential change in the first submodel is the introduction of a single (centralised) counter in the Unify node instead of the many (distributed) counters and state flags of And nodes. Therefore the coupling invariant should be associated to the semantics of the counter. The centralised counter maintains the number of active streams in the ring which is essentially the sum of distributed counters in And nodes.

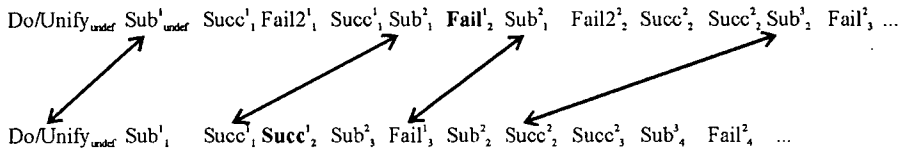

Figure 11: Initial fragment of the commuting diagram for the example runs

Let s_1 the sum of counters in And nodes belonging to the same ring and to the same colour in ASM1 and let s_2 the current value of counter in Unify node of the same colour in ASM2. Let X an initial segment of a run of ASM1 with maximal element m and Y an initial segment of a run of ASM2 with maximal element n. Then $s_1(X)$ means the value of s_1 after performing the steps of X, the meaning of $s_2(Y)$ is similar.

The coupling invariant relates states belonging to initial segments $X - \{m\}$, $Y - \{n\}$ where the value of the counter changes so that

$s_1(X - \{m\}) \neq s_1(X)$, $s_2(Y - \{n\}) \neq s_2(Y)$

and after these steps they are equal

$s_1(X) = s_2(Y)$

then $\sigma(X - \{m\})$ and $\sigma(Y - \{n\})$ are in relation where σ is a projection from segments (sequences of steps) to states.

Let us introduce the following notation: $Token^n_x$ is an event when token of type Token is received at the nth And node in the Unify-And ring with s_1 or $s_2 = x$, whereas Token/Unify means an event caused by a token of type Token at the Unify node. ASM rules are usually guarded by dataflow firing conditions, therefore for the sake of simplicity these events will represent firing the rules. A possible valid run of ASM1 in the Unify-And ring with 3 And nodes is the following list of events:

$Do/Unify_{undef} - Sub^1_{undef} - Succ^1_1 - Fail2^1_1 - Succ^1_1 - Sub^2_1 - Fail^1_2 - Sub^2_1 - Fail2^2_2 - Succ^2_2 - Succ^2_2 - Sub^3_2 - Fail^2_3 - Succ^2_2 - Succ^3_2 - Sub^3_2 - Sub/Unify_3 - Sub^3_3 - Fail^2_4 - Fail2^3_3 - Fail^3_3 - Succ^3_2 - Succ^3_2 - Sub/Unify_2 - Fail^3_2 - Sub/Unify_1 - Succ^3_1 - Fail^3_1 - Sub/Unify_0 - Fail2/Unify_0$

A corresponding run in ASM2 can be obtained by omitting Fail2 events. It means that the embedding graph received and produced tokens in exactly the same timing.

$Do/Unify_{undef} - Sub^1_1 - Succ^1_1 - Succ^1_2 - Sub^2_3 - Fail^1_3 - Sub^2_2 - Succ^2_2 - Succ^2_3 - Sub^3_4 - Fail^2_4 - Succ^2_3 - Succ^3_4 - Sub^3_4 - Sub/Unify_4 - Sub^3_4 - Fail^2_4 - Fail^3_3 - Succ^3_2 - Succ^3_2 - Sub/Unify_2 - Fail^3_2 - Sub/Unify_1 - Succ^3_1 - Fail^3_1 - Sub/Unify_0 - 0/Unify_0$,

where 0/Unify means rule 24.

If both ASM1 and ASM2 were deterministic models, the commuting diagram that is represented in Figure 11 could be easily partitioned by the invariant property showing the equivalence of the two models. But due to the indeterministic nature, the partitioning should be possible for every linearisation of valid runs of ASM1 and ASM2. Therefore, from these strings the general properties must be extracted

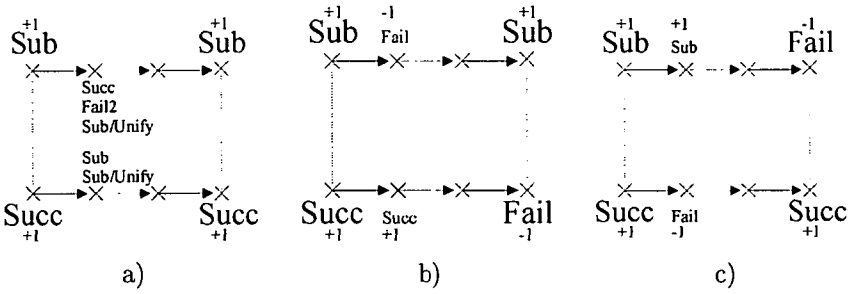

Figure 12: Straight (a) and inverted (b) pairs of states. Inverted pairs (c) is impossible.

omitting the special features of the certain runs. In such a way a general partitioning will be resulted that can be treated as "building blocks" from which all the possible runs can be constructed. If the commutativity for these subdiagrams holds, it holds for all the diagrams that can be constructed of them. (Note that all possible runs can be constructed but not every possible construction is a valid run.)

First it must be shown that from the initial states a related pair of states can be reached. According to the definition, the first Sub event of ASM1 will be related to the Do/Unify event of ASM2 (Figure 13.a.) Hence there are two possibilities: the subgraph connected to the first And node can produce a solution (Figure 13.b) or not (Figure 13.c). Hence at the beginning either a+b or a+c subdiagrams are fixed sequences. The terminating subdiagram d is the only possible terminating sequence and it needs some explanation.

Between the last Fail and the terminating states there cannot be anything except Sub/Unify and Fail2 events. Otherwise, if there were any Sub or Succ events, they would be preceded by their terminating Fail event that is impossible. In such a way the only terminating sequence is subdiagram d in Figure 13. It also shows that the final state can be reached from a related pair of states (namely a Fail-Fail pair).

In ASM1 the relevant states where the counters are modified in any way are represented by Sub and Fail events. Between them any number (including 0) of irrelevant Succ, Fail2 and Sub/Unify event can occur without affecting the sum of counters. In case of ASM2 those events are Succ (except at the last And node in the ring) and Fail with any number of Sub and Sub/Unify (and Succ at the last And node) between them.

In the straight case there are no relevant states between related states (Figure 12.a). It means that at the related states both s_1 and s_2 are incremented or decremented. Therefore, subdiagrams e, f, h, i in Figure 13 can be easily and systematically created. However, it is possible that states are inverted, there are relevant states that are not related, i.e. the next relevant state increments the counter in an ASM and decrements in the other (Figure 12.b, c). It is even possible that there are multiple inverted states. An example for inversion can be seen in bold in Figure 11.

What does it make evident that after an inversion a related pair of states will be reached in Figure 12.b?

Remark 1. Let $\sigma(X)$ and $\sigma(Y)$ be related states. The number of Fail events in X and Y are equal. (Proof: starting from subdiagram a) in Figure 13 and applying straight subdiagrams, the statement holds. Reaching the first inverted pair, the appearance of a Fail event in one ASM guarantees the existence of another Fail event in the other ASM model, because the number of Fail events in the entire runs are equal.)

Remark 2. Succ events are in relation with Sub events that happened later in the run. It is simply a consequence of the fact that Sub events are caused by Succ events (except the first one.)

From these two statements it is true that for the Fail event of ASM1 in Figure 12.b there is another Fail event in ASM2 and for Succ in ASM2 there must be a Sub in ASM1. In such a way the relation holds for the Sub-Fail pair.

On the other hand, the another combination of inverted states is not possible (Figure 12.c). The second Sub event in ASM1 would have been related (through an inversion) to a Succ event that occurs later which is in contradiction with the causality expressed in Remark 2.

As it can be seen, both the start-up and the final stage are of given, fixed types of subdiagrams. There are 3 types of related pairs: Sub-Succ, Fail-Fail and Sub-Fail. From these 9 other types of subdiagrams can be created. In such a way all the valid runs can be constructed from these 13 types of subdiagrams.

For a single diagram, e.g. e) in Figure 13 Schellhorn's theorem states the following. Starting ASM1 from Sub and ASM2 from a related Succ, for every possible successor state in ASM1 there must be a successor state in ASM2 so that the invariant holds again. This is covered by diagrams e), h) and k) in Figure 13. There can be any number of intermediate states, reaching the next Sub/Succ, Fail/Fail or Sub/Fail pair, the relation is true. Hence the proof can be continued starting from the new related states. It is easy to trace the correctness from the very beginning to the end. The commutativity of subdiagrams shows the commutativity of all diagrams constructed of them that means the equivalence of ASM1 and ASM2 according to our definition.

In this step it was assumed that there are no other Unify-And rings in the subgraphs connected to the And nodes in scope. Since it was shown that the Unify-And ring of the modified model behaves like the one in the original Logicflow and hence, from the parent node's point of view there is no difference between a Unit and a Unify node, the equivalence proven here is true even if there are Unify-And rings in the subgraphs attached to the And nodes.

6 Conclusion

In this paper a small part of the design of a distributed parallel Prolog execution model was introduced where Abstract State Machines were applied in the course of development. The outcome of the paper is not a description of a parallel model

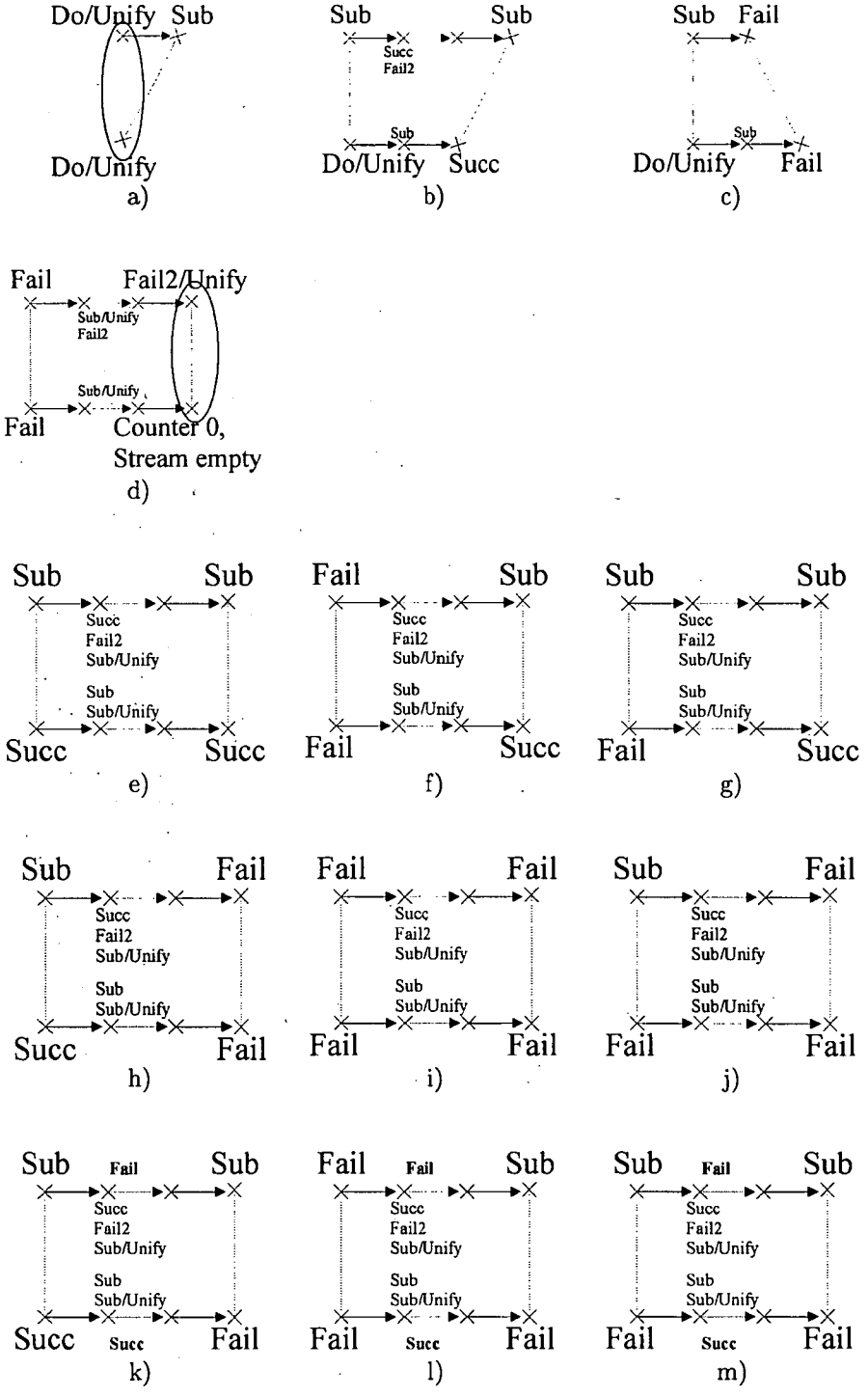

Figure 13: Commuting subdiagrams

ready to be implemented, rather a profitable case study where the application of ASM is demonstrated in different situations.

First, an existing model, Logicflow has been described by ASM1. It is a precise and succinct way of specification that helps to discover the features of the system. Then, this model can be derived to another one by successive minor modifications that are realised by a series of submodels represented by ASMs. In this paper a single step of such modification was introduced. The ASM notation makes clear the scope and the extension of changes.

ASMs are not just a method for description and analysis but provide a framework where models can be compared and their equivalence or inequivalence can be precisely defined and proven. In the current context the equivalence of ASM1 and ASM2 has been proven. The proof method was able to tackle with the distributed and indeterministic nature of dataflow based parallel Prolog models. In summary, experience showed the endowment and efficiency of ASMs (and the related techniques) in system design.

Acknowledgements

Author would gratefully express his appreciate to Prof. Egon Börger who helped with the very first steps in ASMs and gave valuable suggestions and inspiration for the entire work. Prof. Ferenc Vajda and Gábor Dózsa have always been ready for discussion and helped with presenting the work. Earlier stages of the multithreaded Prolog system were investigated in a joint project with Kyushu University where Prof. Makoto Amamiya and Hiroshi Tomiyasu were especially helpful.

References

[1] M.Amamiya, R. Taniguchi: Datarol: A Massively Parallel Architecture for Functional Language. Proc. Second IEEE Symposium on Parallel and Distributed Processing, 1990, pp. 726-735.

[2] Arvind and R.A. Ianucci: Two fundamental issues in multiprocessing. Proc. DFVLR Conf. on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, 1987. pp. 61-88.

[3] E. Börger: High Level System Design and Analysis using Abstract State Machines. In: D. Hutter et al. eds., Current Trends in Applied Formal Methods (FM-Trends 98), LNCS 1641, Springer, pp. 1-43.

[4] E. Börger, D. Rosenzweig: The WAM - Definition and Compiler Correctness. In: C. Beierle and L. Plümer eds, Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence, chapter 2, North Holland, 1994, pp. 20-90.

[5] E. Börger, I. Durdanovic: Correctness of Compiling Occam to Transputer code. Computer Journal, Vol. 39, No. 1, Oxford University Press, 1996, pp. 52-92.

[6] E. Börger: Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, J. Wiederman, editors, Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics, LNCS 1012, Springer, 1995, pp. 236-271.

[7] E. Börger and U. Glässer: Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In: Y. Gurevich and E. Börger, Evolving Algebras Mini-Course, Technical Report BRICS-NS-95-4, BRICS, University of Aarhus, July 1995. http://www.eecs.umich.edu/gasm/papers/pvm.html

[8] Y. Gurevich: Evolving Algebras 1993: Lipari Guide. In: E. Börger ed., Specification and Valdation Methods, Oxford University Press, 1995. pp. 9-36.

[9] Y. Gurevich: Evolving Algebras: An Attempt to Discover Semantics. In: G. Rozenberg, A. Salomaa eds., Current Trends in Theoretical Computer Science, World Scientific, 1993, pp. 266-292.

[10] Y. Gurevich, J.K. Huggins: Equivalence is in the Eye of Beholder. Theoretical Computer Science, Vol. 179, No. 1-2, Elsevier, 1997, pp. 353-380.

[11] Y. Gurevich: May 1997 Draft of the ASM Guide. http://www.eecs.umich.edu/gasm/papers/guide97.html

[12] Y. Gurevich: Sequential Abstract State Machines Capture Sequential Algorithms. ACM Transactions on Computational Logic, Vol. 1, No. 1, July 2000, pp. 77-111.

[13] P. Kacsuk: Execution Models for a Massively Parallel Prolog Implementation. Journal of Computers and Artificial Intelligence, Vol. 17, No. 4, Slovak Academy of Sciences, 1998, pp. 337-364 (part 1) and Vol. 18, No. 2, 1999, pp. 113-138 (part 2)

[14] P. Kacsuk: Distributed Data Driven Prolog Abstract Machine. In: P. Kacsuk, M.J.Wise eds., Implementations of Distributed Prolog. Wiley, 1992. pp. 89-118.

[15] T. Kawano, S. Kusakabe, R. Taniguchi, M. Amamiya: Fine-grain multithread processor architecture for massively parallel processing. Proc. First IEEE Symp. High Performance Computer Architecture (HPCA95), Raleigh, 1995, pp. 308-317.

[16] Zs. Németh, P. Kacsuk: Analysis and Improvement of the Variable Binding Scheme in LOGFLOW. In: I. de Castro Dutra, M. Carro, V. Santos Costa, G. Gupta, E. Pontelli, F. Silva eds., Parallelism and Implementation of Logic and Constraint Programming. Nova Science Publishers, 1999.

[17] Zs. Németh: Abstract machine design on a multithreaded architecture. Future Generation Computer Systems, Vol. 16, No. 6, Elsevier, 2000, pp. 705-716.

[18] Zs. Németh: A Novel Execution Model for LOGFLOW on a Hybrid Multithreaded Architecture. Proceedings of DAPSYS 2000, Kluwer, 2000, pp 117-126.

[19] Zs. Németh: Issues of a Distributed Parallel Prolog System on Hybrid Multithreaded Architectures. PhD Thesis, Budapest University of Technology and Economics, 2001.

[20] B. Robič, J. Šilc, T. Ungerer: Beyond Dataflow. Journal of Computing and Information Technology, Vol. 8, No. 2, University Computing Centre, Zagreb, 2000, pp. 89-101.

[21] G. Schellhorn: Verification of Abstract State Machines. PhD Thesis, University of Ulm, 1999.

[22] D.Sima, T. Fountain, P. Kacsuk: Advanced Computer Architectures. Addison Wesley, 1997.

[23] R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine. Springer, 2001

[24] H.Tomiyasu, T. Kawano, R. Taniguchi, M. Amamiya: KUMP/D: the Kyushu University Multimedia Processor. Proceedings of the Computer Architectures for Machine Perception, CAMP'95, IEEE Computer Society Press, 1995, pp. 367-374.

Appendix A: ASM code for the Logicflow model (ASM1)

1 A DO token on the request.in arc of a Unit node

```
if node(loc(t)) = UNIT & type(t) = DO
then
   extend STREAM by s with
     stream(parent(loc(t)), returnport(loc(t)), colour(t)) := s
     seq i = 1..n
         let θ = mgu(act(t), head(nth(predicate(loc(t)), i)))
         if θ! = nil then
            extend TOKEN by t' with
                   colour(t') := colour(t)
                   loc(t') := parent(loc(t))
                   subst(t') := subst(t)@θ
                   type(t') := SUCC
                   instream(s, t') := true
            endextend
         endif
     endseq
```

```
    loc(t) := parent(loc(t))
    type(t) := FAIL
    subst(t) := {}
    instream(s,t) := true
  endextend
where n = length(predicate(loc(t)))
```

2 A SUB token on the request.in arc of an And node

2a First appearance

```
if node(loc(t)) = AND & type(t) = SUB & mode(loc(t)) = create
then
  extend NODE by n with
    parent(n) := loc(t)
    child(loc(t)) := n
    mode(n) := create
    returnport(n) := reply.in
    predicate(n) := procdef(goal(loc(t)))
    loc(t) := n
  endextend
  if counter(loc(t), colour(t)) = undef
    then
      counter(loc(t), colour(t)) := 1
    else
      counter(loc(t), colour(t)) := counter(loc(t), colour(t)) + 1
  endif
  if and_state(loc(t), colour(t)) = undef
    then
      and_state(loc(t), colour(t)) := open
  endif
  instream(stream(loc(t), request.in, colour(t)), t) := false
  mode(loc(t)) := active
  act(t) := goal(loc(t))
  type(t) := DO
```

6.1 2b Further appearances

```
if node(loc(t)) = AND & type(t) = SUB & mode(loc(t)) = active
then
  if counter(loc(t), colour(t)) = undef
    then
      counter(loc(t), colour(t)) := 1
    else
      counter(loc(t), colour(t)) := counter(loc(t), colour(t)) + 1
  endif
  instream(stream(loc(t), request.in, colour(t)), t) := false
  loc(t) := child(loc(t))
  act(t) := goal(loc(t))
  type(t) := DO
```

3 A FAIL2 token on the request.in arc of an And node

```
if node(loc(t)) = AND
   & type(t) = FAIL2
   & mode(loc(t)) = active
   & card(stream(loc(t), request.in, colour(t))) = 1
   then
      instream(stream(loc(t), request.in, colour(t)), t) := false
      and_state(loc(t), colour(t)) := closed
      if counter(loc(t), colour(t)) = 0 | counter(loc(t), colour(t)) = undef
         then
            instream(stream(next(loc(t)), request.in, colour(t)), t) := true
            loc(t) := next(loc(t))
         else
            TOKEN(t) := false
      endif
```

4 A SUCC token on the reply.in arc of an And node

```
if node(loc(t)) = AND & type(t) = SUCC & mode(loc(t)) = active
then
   if node(next(loc(t))) = UNIFY
      then let port = reply.in
      else let port = request.in
   endif
   if stream(next(loc(t)), reply.in, colour(t)) = undef
   then
      extend STREAM by s with
         stream(next(loc(t)), port, colour(t)) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(next(loc(t)), port, colour(t)), t) := true
   endif
   instream(stream(loc(t), request.in, colour(t)), t) := false
   loc(t) := next(loc(t))
   type(t) := SUB
```

5 A FAIL token on the reply.in arc of an And node when it is open

```
if node(loc(t)) = AND
   & type(t) = FAIL
   & mode(loc(t)) = active
   & card(stream(loc(t), reply.in, colour(t))) = 1
   & and_state(loc(t), colour(t)) = open
   then
      counter(loc(t), colour(t)) := counter(loc(t), colour(t)) - 1
      TOKEN(t) := false
```

6 A FAIL token on the reply.in arc of an And node when it is closed

```
if node(loc(t)) = AND
```

```
& type(t) = FAIL
& mode(loc(t)) = active
& card(stream(loc(t), reply.in, colour(t)))) = 1
& and_state(loc(t), colour(t)) = closed
then
   if node(next(loc(t))) = UNIFY
   then let port = reply.in
   else let port = request.in
   endif
   counter(loc(t), colour(t)) := counter(loc(t), colour(t)) − 1
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   if counter(loc(t), colour(t)) = 0
   then
      instream(stream(next(loc(t)), port, colour(t)), t) := true
      loc(t) := next(loc(t))
   endif
```

7 A DO token on the request.in arc of a Unify node

7a First appearance

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = create
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour_context(loc(t), newcolour) := colour(t)
         subst_context(loc(t), newcolour) := subst(t)@θ
         extend NODE by n1, n2, ...nm with
            node(ni) := AND
            mode(ni) := create
            first(loc(t)) := n1
            prev(n1) := loc(t)
            last(loc(t)) := nm
            next(nm) := loc(t)
            prev(nk) := nk−1
            next(nk) := nk+1
            goal(ni) := nth(body(predicate(loc(t))), i)
            loc(t) := n1
            type(t) := SUB
            subst(t) := θ
            extend STREAM by s with
               stream(first(loc(t)), request.in, newcolour) := s
               extend TOKEN by t' with
                  loc(t') := n1
                  type(t') := FAIL2
                  colour(t') := newcolour
                  instream(s, t') := true
               endextend
               instream(s, t) := true
            endextend
         endextend
         mode(loc(t)) := active
      endextend
```

```
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour(t)) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
   where m = length(body(predicate(loc(t)))), 1 ≤ i ≤ m, 1 < k < m
```

7b Further appearances

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = active
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour_context(loc(t), newcolour) := colour(t)
         subst_context(loc(t), newcolour) := subst(t)@θ
         extend STREAM by s with
            stream(first(loc(t)), request.in, newcolour) := s
            loc(t) := first(loc(t))
            type(t) := SUB
            instream(s, t) := true
            subst(t) := θ
            extend TOKEN by t' with
               loc(t') := first(loc(t))
               type(t') := FAIL2
               colour(t') := newcolour
               instream(s, t') := true
            endextend
         endextend
      endextend
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour(t)) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
```

8 A SUB token on the reply.in arc of a Unify node

```
if node(loc(t)) = UNIFY & type(t) = SUB
then
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   colour(t) := saved_colour
   subst(t) := saved_subst@subst(t)
   type(t) := SUCC
   loc(t) := parent(loc(t))
   if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
   then
```

```
        extend STREAM by s with
            stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
            instream(s, t) := true
        endextend
    else
        instream(stream(parent(loc(t)), returnport(loc(t)), saved_colour), t) := true
    endif
where saved_colour ≡ colour_context(loc(t), colour(t)), saved_subst ≡ subst_context(loc(t), colour(t))
```

9 A FAIL token on the reply.in arc of a Unify node

```
if node(loc(t)) = UNIFY & type(t) = FAIL2 & card(stream(loc(t), reply.in, colour(t))) = 1
then
    let saved_colour = colour_context(loc(t), colour(t))
    instream(stream(loc(t), reply.in, colour(t)), t) := false
    colour(t) := saved_colour
    type(t) := FAIL
    loc(t) := parent(loc(t))
    if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
    then
        extend STREAM by s with
            stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
            instream(s, t) := true
        endextend
    else
        instream(stream(parent(loc(t)), returnport(loc(t)), saved_colour), t) := true
    endif
```

10 A DO token on the request.in arc of an Or node

10a First appearance

```
if node(loc(t)) = OR & type(t) = DO & mode(loc(t)) = create
then
    extend COLOUR by newcolour with
        colour(t) := newcolour
        colour_context(loc(t), newcolour) := colour(t)
        extend NODE by n1, n2 with
            child1(loc(t)) := n1
            child2(loc(t)) := n2
            parent(n1) := loc(t)
            parent(n2) := loc(t)
            returnport(n1) := reply.in1
            returnport(n2) := reply.in2
            if k = 1
            then
                predicate(n1) := car(predicate(loc(t)))
                predicate(n2) := cdr(predicate(loc(t)))
            else
                predicate(n1) := [clause1, ...clausek-1]
                predicate(n2) := [clausek, ...clausen]
            endif
            extend TOKEN by t' with
                loc(t') := n2
                type(t') := DO
```

```
            colour(t') := newcolour
            subst(t') := subst(t)
         endextend
         loc(t) := n1
      endextend
   endextend
   mode(loc(t)) := active
   where clause_i ≡ nth(predicate(loc(t)), i), k = min{i|body(clause_i) ≠ nil}
```

10b Further appearances

```
if node(loc(t)) = OR & type(t) = DO & mode(loc(t)) = active
then
   extend COLOUR by newcolour with
      colour(t) := newcolour
      colour_context(loc(t), newcolour) := colour(t)
      extend TOKEN by t' with
         loc(t') := child2(loc(t))
         type(t') := DO
         colour(t') := newcolour
         subst(t') := subst(t)
      endextend
      loc(t) := child1(loc(t))
   endextend
endextend
```

11 A SUCC token on any of the reply.in arcs of an Or node

```
if node(loc(t)) = OR & type(t) = SUCC
then
   let saved_colour = colour_context(loc(t), colour(t))
   colour(t) := saved_colour
   loc(t) := parent(loc(t))
   if instream(stream(loct(t), reply.in1, colour(t), t) = true
   then
      instream(stream(loct(t), reply.in1, colour(t), t) := false
   else
      instream(stream(loct(t), reply.in2, colour(t), t) := false
   endif
   if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
   then
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(parent(loct(t)), returnport(loc(t)), saved_colour), t) := true
   endif
```

12 A FAIL token on any of the reply.in arcs of an Or node

12a The Or node is in wait1 state

```
if node(loc(t)) = OR
```

```
    & type(t) = FAIL
    & or_state(loc(t), colour(t)) = wait1
    & (∃s ∈ STREAM : instream(s,t) & card(s) = 1)
    then
        or_state(loc(t), colour(t)) := wait2
        TOKEN(t) := false
        STREAM(s) := false
```

12b The Or node is in wait2 state

```
if node(loc(t)) = OR
& type(t) = FAIL
& or_state(loc(t), colour(t)) = wait2
& (∃z ∈ STREAM : instream(z,t) & card(z) = 1)
then
    let saved_colour = colour_context(loc(t), colour(t))
    if stream(parent(loc(t)), returnport(loc(t)), saved_colour) = undef
    then
        extend STREAM by s with
            stream(parent(loc(t)), returnport(loc(t)), saved_colour) := s
            instream(s,t) := true
        endextend
    else
        instream(stream(parent(loct(t)), returnport(loc(t)), saved_colour), t) := true
    endif
    STREAM(z) := false
    colour(t) := saved_colour
    loc(t) := parent(loc(t))
```

13 A DO token on the request.in arc of an undefined node (child nodes of And and Or nodes are undefined)

```
if node(loc(t)) = undef & type(t) = DO
then
    if length(predicate(loc(t)) = 1
    thenif body(clause_1) ≠ nil
        then
            node(loc(t)) := UNIFY
        else
            node(loc(t)) := UNIT
        endif
    elseif ∀i : body(clause_i) = nil
        then
            node(loc(t)) := UNIT
        else
            node(loc(t)) := OR
        endif
    endif
    where clause_i ≡ nth(predicate(loc(t)), i)
```

14 A DO token on a Query node

```
if node(loc(t)) = QUERY & type(t) = DO
then
```

```
extend NODE by n with
    child(loc(t)) := n
    parent(n) := loc(t)
    predicate(n) := procdef(act(t))
    mode(n) := create
    returnport(n) := reply.in
    loc(t) := n
endextend
```

15 A SUCC token on a Query node

```
if node(loc(t)) = QUERY & type(t) = SUCC
then
    TOKEN(t) := false
```

16 A FAIL token on a Query node

```
if node(loc(t)) = QUERY & type(t) = FAIL & card(stream(loc(t), reply.in, colour(t))) = 1
then
    TOKEN(t) := false
    STREAM(stream(loc(t), reply.in, colour(t))) := false
```

Appendix B: ASM code for the modified Logicflow model (ASM2)

```
ancestor : NODE → NODE
counter : NODE × COLOUR → INT
```

17 A DO token on the request.in arc of a Unit node

Same as Rule 1

18 A SUB token on the request.in arc of an And or Last_And node

6.2 18a First appearance

```
if node(loc(t)) = (AND|LAST_AND) & type(t) = SUB & mode(loc(t)) = create
then
    extend NODE by n with
        parent(n) := loc(t)
        child(loc(t)) := n
        mode(n) := create
        returnport(n) := reply.in
        predicate(n) := procdef(goal(loc(t)))
        loc(t) := n
    endextend
    instream(stream(loc(t), request.in, colour(t)), t) := false
    mode(loc(t)) := active
    act(t) := goal(loc(t))
    type(t) := DO
```

18b Further appearances

```
if node(loc(t)) = (AND|LAST_AND) & type(t) = SUB & mode(loc(t)) = active
then
   instream(stream(loc(t), request.in, colour(t)), t) := false
   loc(t) := child(loc(t))
   act(t) := goal(loc(t))
   type(t) := DO
```

19 A SUCC token on the reply.in arc of an And node

```
if node(loc(t)) = AND & type(t) = SUCC & mode(loc(t)) = active
then
   if stream(next(loc(t)), request.in, colour(t)) = undef
   then
      extend STREAM by s with
         stream(next(loc(t)), request.in, colour(t)) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(next(loc(t)), request.in, colour(t)), t) := true
   endif
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   counter(ancestor(loc(t), colour(t)) := counter(ancestor(loc(t), colour(t)) + 1
   loc(t) := next(loc(t))
   type(t) := SUB
```

20 A SUCC token on the reply.in arc of a Last_And node

```
if node(loc(t)) = LAST_AND & type(t) = SUCC & mode(loc(t)) = active
then
   if stream(next(loc(t)), reply.in, colour(t)) = undef
   then
      extend STREAM by s with
         stream(next(loc(t)), reply.in, colour(t)) := s
         instream(s, t) := true
      endextend
   else
      instream(stream(next(loc(t)), reply.in, colour(t)), t) := true
   endif
   instream(stream(loc(t), reply.in, colour(t)), t) := false
   loc(t) := next(loc(t))
   type(t) := SUB
```

21 A FAIL token on the reply.in arc of an And or Last_And node

```
if node(loc(t)) = (AND|LAST_AND)
   & type(t) = FAIL
   & mode(loc(t)) = active
   & card(stream(loc(t), reply.in, colour(t))) = 1
   then
      counter(ancestor(loc(t)), colour(t)) := counter(ancestor(loc(t)), colour(t)) - 1
      TOKEN(t) := false
```

22 A DO token on the request.in arc of a Unify node

22a First appearance

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = create
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour_context(loc(t), newcolour) := colour(t)
         subst_context(loc(t), newcolour) := subst(t)@θ
         counter(loc(t), newcolour) := 1
         extend NODE by n1, n2, ...nm with
            if i < m then
                        node(ni) := AND
            else
                        node(ni) := LAST_AND
            endif
            mode(ni) := create
            ancestor(ni) := loc(t)
            first(loc(t)) := n1
            prev(n1) := loc(t)
            last(loc(t)) := nm
            next(nm) := loc(t)
            prev(nk) := nk-1
            next(nk) := nk+1
            goal(ni) := nth(body(predicate(loc(t))), i)
            loc(t) := n1
            type(t) := SUB
            subst(t) := θ
            extend STREAM by s with
               stream(first(loc(t)), request.in, newcolour) := s
               instream(s, t) := true
            endextend
         endextend
         mode(loc(t)) := active
      endextend
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
where m = length(body(predicate(loc(t)))), 1 ≤ i ≤ m, 1 < k < m
```

22b Further appearances

```
if node(loc(t)) = UNIFY & type(t) = DO & mode(loc(t)) = active
then
   let θ = mgu(act(t), head(predicate(loc(t))))
   if θ! = nil
   then
```

```
      extend COLOUR by newcolour with
         colour(t) := newcolour
         colour(loc(t), newcolour) := colour(t)
         subst(loc(t), newcolour) := subst(t)@θ
         counter(loc(t), newcolour) := 1
         extend STREAM by s with
            stream(first(loc(t)), request.in, newcolour) := s
            loc(t) := first(loc(t))
            type(t) := SUB
            instream(s, t) := true
            subst(t) := θ
         endextend
      endextend
   else
      extend STREAM by s with
         stream(parent(loc(t)), returnport(loc(t)), colour) := s
         type(t) := FAIL
         instream(s, t) := true
         loc(t) := parent(loc(t))
      endextend
   endif
```

23 A SUB token on the reply.in arc of a Unify node

Same as Rule 8

24 The counter of a Unify node is 0

```
if (∃Node, Colour : Node ∈ NODE, Colour ∈ COLOUR) : node(Node) = UNIFY
   & counter(Node, Colour) = 0
   & card(stream(Node, reply.in, Colour)) = 0
   then
      let saved_colour = colour_context(Node, Colour)
      extend TOKEN by t' with
         colour(t') := saved_colour
         type(t') := FAIL
         loc(t') := parent(Node)
         if stream(parent(Node), returnport(Node), saved_colour) = undef
         then
            extend STREAM by s with
               stream(parent(Node), returnport(Node), saved_colour) := s
               instream(s, t') := true
            endextend
         else
            instream(stream(parent(Node), returnport(Node), saved_colour), t') := true
         endif
      endextend
```

25 A DO token on the request.in arc of an Or node

25a First appearance

Same as Rule 10a

25b Further appearances

Same as Rule 10b

26 A SUCC token on any of the reply.in arcs of an Or node

Same as Rule 11

27 A FAIL token on any of the reply.in arcs of an Or node

27a The Or node is in wait1 state

Same as Rule 12a

27b The Or node is in wait2 state

Same as Rule 12b

28 A DO token on the request.in arc of an undefined node (child nodes of And and Or nodes are undefined)

Same as Rule 13

29 A DO token on a Query node

Same as Rule 14

30 A SUCC token on a Query node

Same as Rule 15

31 A FAIL token on a Query node

Same as Rule 16