

A NEW DIALECT OF SOFL: SYNTAX, FORMAL SEMANTICS, AND TOOL SUPPORT

by

Johan Sarel van der Berg

Submitted in partial fulfilment of the requirements for the degree

Master of Science (Computer Science)

in the

Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering, Built Environment and Information Technology
University of Pretoria, Pretoria

December 2018

Summary

A NEW DIALECT OF SOFL: SYNTAX, FORMAL SEMANTICS, AND TOOL SUPPORT

by

Johan Sarel van der Berg

Supervisor:	Prof. S. Gruner
Department:	Faculty of Engineering, Built Environment and Information Technology
University:	University of Pretoria
Degree:	Master of Science (Computer Science)
Keywords:	Process algebra, Formal modeling, Formal semantics, mCRL2, Restricted dialect, SMT-Solver, Structured Object Orientated Formal Language, SOFL, Tool support, Z3

Structured Object Orientated Formal Language (SOFL) is a formal method design methodology that combines data flows diagrams and predicates in order to describe processes that can be refined. This methodology creates a very versatile method of describing a system, which system properties can be proven rigorously. Data flows are grouped by ports that define from which data flows data can be consumed or on which flows data can be generated. For predicates, Logic of Partial Functions (LFP) are used; and an undefined element that is also used to indicate if a data flows do not contain any data.

Over time SOFL “evolved organically” and a number of features were added: usability was the main consideration for a feature being added. For a formal language to be useful there must be no uncertainty of a specific design’s meaning. With SOFL, there is a possible contradiction between the requirement that a process’s precondition must be true when the process fire, and the fire rules. This contradiction is due to the use of LFP.

Semantics (the meaning) of SOFL was not always updated to keep track of the changes made to SOFL which resulted in an outdated and incomplete semantic. The incompleteness of the semantics is a significant factor motivating the work done in this dissertation.

In this dissertation, a dialect of SOFL is created to define a semantic. Not all the elements of SOFL are added in order that a simpler semantic can be defined. Elements that were removed include:

- LFP,
- Classes, and
- Non-deterministic broadcast nodes.

Semantics of the dialect is created by a two-step process: firstly, an intuitive understanding of the dialect is created, and secondly, both static and dynamic semantics are defined by means of translations.

A translation is a mapping from the dialect to a formal language that describes a certain aspect of the dialect. Static semantics defines the meaning of the elements that are “fixed” in their state: SMT-LIB is used as the target language to describe the static semantics of the dialect. Dynamic semantics describes how an element in a design changes over time: the process algebra mCRL2 is used as the formal language which describes the dynamic behaviour of the dialect.

The SMT-Solver Z3 and tools included in mCRL2 are used to analyse the translation of the dialect. Use of these tools allows properties that are necessary for a design to have a well defined meaning, to be proven. Properties that can be proven include: a process can fire, a process can fire an infinite number of times, and a predicate that described a property.

An Eclipse plug-in is created so that translation is not required to be done manually. After a design is translated the tools Z3 and mCRL2 are run using script files and the results of the analysis are displayed on the screen. The desired properties could be proven but for a moderate size design, but as the size of the design increased the analysis of the translation could not be completed due to computational problem. Usability of the tool can be improved by not only using a textual representation of a design, but also visual representations as in SOFL.

As a result, properties that are necessary for a design to have a well-defined meaning, can be proven using these tools.

Acknowledgement

During the course of the dissertation there were several discussions with the following experts that were especially helpful: Stefan Gruner (Supervisor), Shaoying Liu, Markus Roggenbach, Jan Friso Groote, Stefan Blom, Jaco v.d. Pol, Christian Dietrich and the anonymous reviewers that provided their insightful comments on my and Prof Guner's paper draft for SOFL+MSVL'18 workshop. The comments received after the examination from the external reviewers were also valuable.

Contents

List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 History of SOFL	1
1.1.1 Origins	1
1.1.2 Structured design	2
1.1.3 Object-Orientation	2
1.1.4 Parallelism and Concurrency	2
1.1.5 Verification	3
1.1.6 Semantics	3
1.2 Problem Statement	4
1.3 Method	5
1.4 Contributions	5
1.5 Modification	6
1.5.1 Refinement and Hierarchical Structures	6
1.5.2 Logic Usage in Specification	7
1.5.3 Data Flows Diagrams	9
1.5.4 Data Types	10
1.6 Associated publication	10
1.7 Outline	10
2 Related Work	12
2.1 Rigorous Review	12
2.1.1 Properties	13
2.1.2 Method	13
2.1.3 Comments	14
2.2 Specification Verification	14
2.2.1 Properties	15
2.2.2 Method	15
2.2.3 Comments	15
2.3 Semantics	15
2.3.1 Static Semantics	15
2.3.2 Dynamic Semantics	16
2.3.3 Revisit of Refinement	16

2.3.4	Comments	17
2.4	Summary	18
3	Description of SOFL Dialect	19
3.1	Syntax	19
3.1.1	Module	19
3.1.2	Behaviour	20
3.1.3	Processes	25
3.1.4	Expression and Function	26
3.1.5	Syntax Differences	26
3.2	Semi-Formal Semantics	27
3.2.1	Program Graph	28
3.2.2	Predicate Transforms	30
3.2.3	Refinement	35
3.2.4	Action Systems	38
3.3	Verification	41
4	Static Semantics	43
4.1	Type Checking	43
4.1.1	Basic Types	43
4.1.2	Data Flow Types	43
4.2	Scope and Variable Definitions	45
4.3	Motivation for Use of SMT-Solver	46
4.4	SMT-LIB Usage	47
4.5	Translations	49
4.5.1	Refinement	49
4.5.2	Invariants and Initialisation	52
4.5.3	Node State Space Transition	53
4.5.4	Execution Path Approximation	56
5	Dynamic Semantics	60
5.1	Firing Rules	60
5.2	Motivation for Process Algebra	62
5.3	mCRL2 usage	64
5.4	Process Algebra Description	68
5.5	LTS of Action System	72
5.6	Modal Logic	75
5.6.1	μ -Calculus formulæ	77
5.6.2	Property Formulæ	78
5.7	Analysis using mCRL2	79
6	Developed Tool and Evaluation	83
6.1	Developed Tool	83
6.1.1	Workspace	84
6.1.2	Xtext Parsing	85
6.1.3	Xtext Semantics Verification	85
6.1.4	Translation Generation	86

6.1.5	Final Analysis Step	86
6.2	Examples	88
6.2.1	Small Example	88
6.2.2	Large Example	90
6.2.3	Multi-port Example	91
6.2.4	Producer Delivery Example	91
6.3	Time Analysis	94
6.4	Correctness of Tool	97
7	Future Work and Discussion	98
7.1	User Experience	98
7.2	Language usability	98
7.3	Correctness of Tool	98
7.4	Specifying Properties	99
7.5	Semantics	99
7.6	Thoughts on SOFL	99
7.6.1	Non-determinism	99
7.6.2	Classes and Data Types	100
7.6.3	Semantics	100
7.7	Conclusion	102
	Bibliography	103
A	Grammar	109
A.1	Module and CDFD Syntax	109
A.2	Behaviour	110
A.3	Process and Port Syntax	111
A.4	Type Syntax	112
A.5	Expression and Function Syntax	113
B	Example Specification	116
B.1	Small Example	116
B.1.1	Specifications	116
B.1.2	mCRL2 Description	119
B.1.3	Results	133
B.2	Multi-port Example	136
B.3	Producer	136

List of Figures

1.1	SOFL design with deadlock	4
2.1	Related work	12
2.2	A task review tree of an expression	14
3.1	Example of a CDFD	21
3.2	A conditional structure representation.	23
3.3	Merging and separating structure representation	24
3.4	A broadcast structure representation	24
3.5	A graphical representation of a data store	25
3.6	A graphical representation of a process	26
3.7	Example of program diagram	29
3.8	Predicate transform diagram	32
3.9	Predicate transform diagram and “execution” paths	33
3.10	Process refined by CDFD	36
3.11	Predicate transform diagram during refinement	36
3.12	CDFD verification order	42
4.1	The pre image of an action a_i	57
5.1	Event order in mCRL2 description	61
5.2	Tree of behaviours	62
5.3	Tree of process algebra	66
5.4	ACP processes used to specify a CDFD	68
5.5	Small example’s LTS showing a CDFD’s input/output	81
5.6	Small example’s LTS showing data store access and nodes that fire	82
6.1	The elements of the created plug-in	83
6.2	Editor for the syntactic and semantic analysis of SOFL specifications	87
6.3	CDFD of small example	88
6.4	CDFD of large example	91
6.5	State transition diagram of a delivery system	92
6.6	CDFD of producer example	93

List of Tables

1.1	Operation tables for LPF	8
5.1	Description of actions used in the ACP description	75
5.2	Actions in LTSs	76
B.1	Analysis of SMT-LIB results	135
B.2	Results from evaluating μ -Calculus formulæ	136

Chapter 1

Introduction

Design of a system is one of the first steps before a system can be implemented, and assurance that a design is correct is an important aspect of a good design. A design language is used to assist in designing a system. A “good” design language will provide assistance by only allowing designs that have a well-defined semantics.

Structured Object Orientated Formal Language (SOFL) is a formal method design methodology that provides limited assistance. Throughout this dissertation the description given in [46] will be used as defining the SOFL “standard”. Familiarity with SOFL will be assumed. This dissertation will focus on a dialect of SOFL and the semantics of the dialect.

1.1 History of SOFL

SOFL has a long history and “evolved organically” as new features were added. This section gives an overview of the development of SOFL.

1.1.1 Origins

The foundation of SOFL is rooted in Formal Requirement Specification Method (FRSM) [40,41] which provided the basis of data flow diagrams and processes. FRSM is a structured system analysis method based on DeMarco data flow diagrams using data processing units called processes and structures – both are also called nodes. These are used to control the flow of data in the diagrams (like conditional structures). Nodes are related to transition in Petri-Nets [31] and have the same properties in that only some input data can be used to produce output on some of its output data flows. In FRSM and SOFL these groupings of data flows are called ports, and each node has a set of input and output ports.

A formal methods element is added for each process by defining the pre and post conditions, and allowing a process to be refined (decomposed) using a Condition Decision Flow Diagram (CDFD). The predicates are defined using the notation of the Vienna Development Method (VDM) [32], which allows the use of undefined elements that are used when a data flows contains no data and a sub-expression cannot be evaluated.

1.1.2 Structured design

A top-down approach to designing systems and applying SOFL in the design was considered in [47] and used in the phase of SOFL where the client is engaged to identify the requirements of the system. Moving from the top to the bottom, a hierarchical structure is created using CDFD, which adds additional information at each level. Each level needs to be defined with enough abstraction so that lower levels can be adapted to the solution without needing too many changes to the levels above. The lowest level is much more formal and the focus is on how to perform the needed actions instead of defining the functionality that is required from the system.

Whenever the hierarchy focuses on how to perform the needed actions, the approach is called a scenario-based method and a bottom-up approach is taken. This approach is usually much more formal since each node used in a CDFD is already defined and the requirements of each node need to be satisfied whenever the nodes are used.

A tool was also developed in [47] as an aid in using both top-down and bottom-up design processes. As an example of how to implement these ideas, a train ticket purchasing system was considered.

1.1.3 Object-Orientation

In [51] an approach to combining Unified Modeling Language (UML)¹ classes into SOFL is considered by viewing each SOFL process as an instance of a class. In order to determine how useful this integration is, top-down and bottom-up design strategies were considered. It was found that integration did not solve all the problems of a conventional object-oriented design.

The use of template classes as a data type was introduced in [52] and new firing rules are put forward. The new execution rules are simplified by only requiring that all input flows contain data of a node, for the node to be enabled. This is different from the firing rules defined in [46] where the requirement is for input data flows connected to the same port. The parallelism of general data flow diagrams [17] was also made explicit by allowing more than one process to execute at the same time. This parallelism requires that data stores are accessed so that the value they store remains consistent, read/write mutual exclusion.

1.1.4 Parallelism and Concurrency

Data flow diagrams define implicit parallelism [17] by allowing different computational units to consume data (when all the data needed to perform the computation is available) and execute the nodes that consume data in parallel. A CDFD allows processes to be implemented on different computational units. This was formalised for SOFL in [14]. By explicitly indicating the concurrency in a design, the implementer is given additional guidance when an implementation is created from the design.

Regions were defined for a CDFD in order to group processes together to indicate that their “execution” is independent of the other processes in the same CDFD. Total

¹<http://www.uml.org/>

independence is not required since, in addition to data flows, message passing is defined which allows processes in different regions to communicate.

1.1.5 Verification

Specification-Based Testing (SBT) [62, 63] is where data values are created that satisfy a specification and those values are used to verify that an implementation functions correctly. For SOFL, SBT generated data is used to perform a “spot check” simulation in order to determine if a specification is correct.

Generation of data for a SOFL design uses a predicate based method [55], Functional Scenario (FS) [36], or a combination of the two methods. A predicate based approach is where the structure of predicate is used to decompose into sub-expressions. The “intended” input data values are determined so that different grouping of data values satisfy the different sub-expressions. The idea is that the generated data will test functionality captured by the predicates of a specification during simulation. FS is where processes in a CDFD are grouped together and the input and output values of these groupings are analysed. The data values, used as intermediate values between the processes, are not considered. This approach gives abstractions of processes used in CDFD which are easier to verify than the complete CDFD.

Simulations used to “execute” a specification that are based on data values that satisfy input and output predicates, are given in [42]. An improvement [38] that uses a visual approach to show the simulations uses Message Sequence Chart (MSC) [29] sequential diagram and FS to generate data used in the simulation.

Proving properties that are a logical consequence of a design makes use of Review Task Tree (RTT) [45]. RTT is a visual representation of a predicate that gives a tree of the expression structure. This is used to assist in proving the property expressed by the predicate. This technique was applied to an ATM example [43] and tool support was developed in [39].

1.1.6 Semantics

Semantics define the meaning of the specification language’s syntax. Semantics is usually defined very formally by using precisely defined concepts and a logical framework. Usually semantics is divided into two parts; static semantics is used to define elements that do not change, and dynamic semantics is used to define how elements change over time. Static [19, 20] and dynamic [30] semantics of SOFL are defined using Objective-Z and First Order Logic (FOL), respectively. Other work done on semantics of SOFL is [48], where the semantics of pre/post condition are revisited. In this case, each refinement step is required to be a valid program step, thereby adding an additional constrain on what defines a valid refinement.

A simplification of SOFL was considered in [25, 26, 75] and called Predicate Data Flow Diagram (PDFD) where each process is allowed to have only one input and output port. This semantics is not directly applicable to SOFL but gives a “gentle” introduction to similar elements used in SOFL.

1.2 Problem Statement

Structured Object Orientated Formal Language (SOFL) “evolved organically” over the years since 1995 [57]. The semantics was not updated with every change made to SOFL. If a semantic was given, it was usually not integrated with existing semantics and it was discussed separately from previous semantics. Having one semantics that describes all aspects using the same “language”, helps to understand the semantics as well as to identify elements that need to be improved.

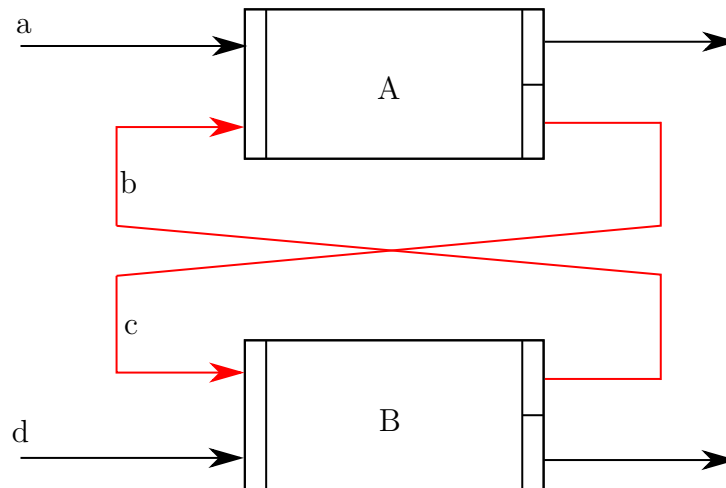


Figure 1.1: A valid design in SOFL design that deadlocks intermediately

The verification of specifications is not a trivial task, therefore, considerable work was done before it could be determined if a SOFL design was valid. A deadlock-free design is an important property of a design, and Figure 1.1 shows a CFD where the process is immediately in a deadlock state. This is not a valid design because it does not satisfy [46, Def 30] since the design cannot produce any output, and the amount of work needed to verify increases exponentially as the size of the design increases. Currently only simulations of SBT are defined to verify properties of a SOFL specification, however, more formal methods will provide more confidence in a design.

This study aims to provide a semantic for a dialect of SOFL and define how a specification is verified using this semantic. Part of the aim will be to determine how effective the verification process function is. The goals are summarised as follows:

1. Obtain a dialect of SOFL that is minimal for defining the semantics.
2. Create a semantics for the dialect of SOFL which satisfy the following properties:
 - (a) A semi-formal unified view exists for both static and dynamic semantics.
 - (b) A static semantics of the dialect.
 - (c) A dynamic semantics of the dialect.

3. Decrease the workload, or obtain more information with the same workload, as designers already have a large workload solving the formulæ needed to prove a design is valid.

1.3 Method

During the research, areas of interest were identified where certain techniques were applied. The areas and the techniques are described below:

Design language. A dialect is created that is based on SOFL by removal of elements or modifications. The purpose of these changes is to define a semantics, therefore reasons related to the semantics are used.

Approximation of semantics. For the semantics, created approximations are used. The reason for using approximations is that not all the information is available² when the specific type of semantics is described, and considering complete state information will result in a scenario that is not always computable during analysis.

Unified view. A general idea of the meaning of a specification is needed by both the designer and the implementer of the specification. A semi-formal unified view creates a common understanding between the designer, implementer, and the semantics of the dialect.

Translations. Translation are used to create mappings from the unified view to either Satisfiability Modulo Theories (SMT)-LIB or a process algebra. SMT-LIB is used to define the static semantics since information of the static element in the design is used in creating the translation. The process algebra is used to define the dynamic semantics since the fire rules of nodes are used in the translation; here the process algebra is used to describe the behaviour during “execution” of a CDFD.

Proving properties. The model created by the semantics for each design allows designs that are not considered to be valid. These larger than necessary models are used so that it can be determined if a design contains unwanted properties. This approach was taken to prevent “forcing” a model to satisfy certain properties, by using the minimum amount of information when creating the translations.

Automation. In order to prevent an increased workload for the designer, automation is used to perform the translation and checks where possible. The tools, Eclipse and Z3 [61], and tool-set micro Common Representation Language 2 (mCRL2) [16], are used during the translations and proving of properties.

1.4 Contributions

A dialect is created so that the semantics of the dialect is “smaller” than the semantics of SOFL and therefore more manageable when analysing. This dialect does not considerably

²This is due to the separation between static and dynamic semantics.

change the meaning of a design and gives the option to extend it to create a language that has equivalent, if not better, descriptive power than SOFL.

SOFL allows a number of specification that are not valid³ and provides very limited support to detect invalid specifications. The designer needs to use labour-intensive methods to analyse a design. This requires that the designer is an “expert” in SOFL, logic, and concurrent systems, to name but a few. By automating some verification, the usability of the created dialect of SOFL was improved, and therefore slightly less expert knowledge is needed to use the dialect.

From the literature review, no other case was found where the same techniques as here were applied to define the semantics of SOFL. An approximation is created for the semantics that allows a design to have properties that are not allowed. Since a model can be created, the properties can be tested formally. This is very close to the approach taken in [46], but they eliminate design as soon as a property is not satisfied. This early elimination will not allow the model to be created, as a result, the properties need to be verified manually.

The theory applied to define a semantics is older than the semantics that exist for SOFL. Independent development of the theory resulted in tools being created that are based on the theory used in this dissertation, and some of these tools are used here to perform formal verification.

The new semantics requires that all nodes must terminate, and that all nodes and CDFDs generate data. Termination of nodes allows the use of the idea of “total” correctness, which is also the basis for the “Correctness by Construction” approach. The requirement that all nodes generate data is needed by the description in a process algebra, but also gives the added advantage of finer control of how processes fire. Instead of allowing a number of nodes that consume data to result in termination of a CDFD, a subset of these nodes can be defined and each subset can independently result in the CDFD terminating.

Modal formulæ can now be defined to describe dynamic properties of a CDFD. This allows the use of a program to verify the formulæ, and results in a lighter workload for the designer. The translations used to create the formulæ can be improved, and important problems like detecting deadlock is still open for designs of moderate size.

1.5 Modification

Comparing the newly created dialect with original SOFL, a number of modifications exist, and this section describes the modifications that have the most impact. Modifications are also made to the syntax and the syntax of the dialect is given in Appendix A.

1.5.1 Refinement and Hierarchical Structures

A process in a CDFD is refined by defining a new CDFD so that the process is described in more detail. This refinement of processes creates a hierarchical structure of CDFDs. SOFL uses a hierarchical structure for one of two purposes:

1. To capture the requirement of the system being designed. A top-down approach.

³For example see Figure 1.1

2. To create a more formal specification where the focus is how to realise the requirements. A bottom-up approach.

Both these uses create a hierarchical structure where a lower level in the structure adds additional information to the specification.

The use of a hierarchical structure in this dissertation will be closer to the description given in item 2 (in the above list) during analysis of the semantics. This allows a more formal approach as well as working with fewer “undefined” elements when a specific CDFD is analysed. During analysis, it is assumed that each process in the CDFD is implementable and the higher in the hierarchical structure the CDFD is, the “greater” this assumption. By starting analysis at the end point in the hierarchical structure, the designer adds all the information possible and any assumptions about processes in the CDFD being analysed. This will create an analysis with less assumption compared to starting the analysis from the top. This approach is very similar to “Correctness by Construction” [35] and [18].

A CDFD sometimes requires the use of additional input and output data flows that do not exist in the process being refined. These new data flows are introduced in an ad-hoc manner. The data type transported by the data flows must be derived from the port to which the data flows is connected. No information is provided about which groups of the input data flows must consume input, or which groups of output data flows are allowed to generate data simultaneously. The dialect requires that each CDFD has input and output ports like any process.

1.5.2 Logic Usage in Specification

Logic is used in SOFL to define state spaces of data values that are allowed to be assigned to either data store or data flows. This section compares Logic of Partial Functions (LPF) and FOL before it is decided in favour of FOL, after which, other modifications related to the application of logic are discussed.

Case for use of LPF

SOFL uses LPF [27, 33] for defining any predicate. LPF extends FOL by allowing three “elementary values”: “true”, “false” and “undefined” value (\perp) and evaluation of conjunction, disjunction, negation and implication are shown in Tables 1.1. An application of LPF is to define the pre and post condition of a function where the function might not terminate when the pre condition is not satisfied. This allows evaluation of a formula *pre condition* \rightarrow *post condition* as part of a larger formula without the need to consider the special case where the function does not terminate. A more elaborated description of LPF in formal specifications is given in [23, 34]. This is a very specific application, and fortunately it is not applicable to SOFL when a process’ pre and post condition are considered, since termination of processes is guaranteed.

There is merit in using LPF in SOFL when formulæ are created that contain variables that refer to “complex” data types, i.e., lists. Consider the evaluation of a function $lst(0)/len(lst) > 10$ where *lts* is an empty list, thus the value of *lst*[0] is undefined, as well as division by a zero value. It can be seen that a “pre condition” of the function is

\wedge	T	\perp	F
T	T	\perp	F
\perp	\perp	\perp	F
F	F	F	F

(a) Operation table of conjunction in LPF

\vee	T	\perp	F
T	T	T	T
\perp	T	\perp	\perp
F	T	\perp	F

(b) Operation table of disjunction in LPF

\neg	
T	F
\perp	\perp
F	T

(c) Operation table of negation in LPF

\longrightarrow	T	\perp	F
T	T	\perp	F
\perp	T	\perp	\perp
F	T	T	T

(d) Operation table of implication in LPF

Table 1.1: Operation tables for LPF

that the list must not be empty, and avoiding the case of undefined values the function should be written as

$$\left(\text{len}(lst) > 0 \longrightarrow \left(\frac{lst(0)}{\text{len}(lst)} > 10 \right) \right) \vee \text{false}.$$

and only evaluate $lst(0)/\text{len}(lst) > 10$ when $\text{len}(lst) > 0$ is true.⁴ By using LPF, “tricks” as above are not necessary to express the above formula. The difficulty in defining a formula where a sub-expression is undefined (like $lst[0]$ in the above case) is a general problem in formal specification and not specific to SOFL. Since the purpose is to understand SOFL better, this scenario is not sufficient to motivate the use of LPF in the dialect of SOFL.

Case for use of FOL

SOFL uses one formula to define a pre condition and another to define a post condition of a process. These formulæ reference the connected data flows and data stores that the process has access to. Whenever a formula uses a data flows that does not contain a value, the value “undefined” is assigned to the variable associated with the data flows. However useful this is for the designer, it creates “hidden” relations between input (or output) ports of a process, which can lead to unexpected behaviour.

For example: Consider a process with two input ports where the first port is connected to a data flow a and the second data flow connected to data flows b and c , where the pre condition is given by $(a \geq 0 \wedge \text{bound}(b)) \vee (b \geq 10)$. The process will fire when there are data on a or on b and c , but the pre condition will only evaluate to true if there is data on b . Since the pre condition must evaluate to true when the process fires, for the design to be valid it must always be the case that, if there is data on a there must be data on b . Thus, that there must be data on a and b contradicts the idea that a port groups related data flows together and data is consumed from these data flows grouped together. This is an example of a process that is not useful in a specification as its meaning is questionable. This is due to the use of “undefined” values in the formula that cause the behaviour.

⁴This is non-standard type of formula evaluation in LPF and FOL, also see [34]

Use of LPF addresses a general problem of when pre and post conditions are used to define a specification of a function, but also allows the definition of a process so that a “hidden” relationship between its input ports is created. The focus of this dissertation is to define a semantics for a dialect of SOFL and not to solve general problems in formal methods. Therefore, it is considered more important to address the “hidden” relationship that can exist between input ports of a process.

State Space

Use of predicate transform (i.e., the weakest precondition) is central to the idea of proving correctness of data being transformed, see [35] and [18]. These ideas are indirectly used in SOFL, but these concepts were used directly when the semantics for the dialect was defined. The state of CDFD is changed by a predicate transform and therefore no predicate is allowed to change the state of a CDFD. State changes are realised by a transition caused by process and a transformation defined by the combined effect of data flow through a CDFD.

The use of explicit specifications will increase the complexity of defining state spaces. Removal of explicit specifications avoids this complexity and the use of a language like Guard Command Language (GCL) [18], and allows the use of CDFD to define how state spaces change. A process is now only allowed to specify state changes by its pre and post condition, and refinement by a CDFD adds additional information on how the state changes are realised.

Initialising a Module

SOFL defines initial values of data stores by defining explicit values, where these values are computed by evaluating expressions. Since a fixed value is just a state space that only allows that value, there is no reason that a state space cannot be used to define the initialisation of a CDFD.

Restricted Usage of Logic

In [43] an invariant of module was used to reference data flows. The use of predicate in such a manner is seen as very informal, since there is no semantics defined to describe how the evaluation of the predicate interacts with the semantics.

The approach taken in defining the semantics of the dialect is to create a graph from a CDFD, and the graph is used as the Kripke structure [10] of a modal logic. This allows modal formulæ to be firmly rooted in the semantics of the dialect, and allows properties of a data flow diagram to be investigated by using modal formulæ.

1.5.3 Data Flows Diagrams

The most notable change to the data flow diagram is that there is no support for a graphical representation of a CDFD. This negatively affects the ease of use from a designer’s perspective, but does not prevent the problem statement from being addressed.

Non-deterministic broadcast nodes are not allowed in the dialect, as the semantic of a non-deterministic broadcast requires knowledge of the “future”. Knowledge of the “future”

would increase the complexity of the semantics and translation and it was decided that the increase in complexity is not worthwhile.

Shadow data flows are introduced too, so that ports not connected to data flows (as defined in classical SOFL) are now connected to shadow flows. They are treated the same as active data flows when determining if a node can fire. Passive data flows influence the process that can fire by enabling or disabling the process. The use of a passive data flow to prevent a node from consuming data is seen as totally foreign to the basic idea of a data flow diagram; each computational unit in a data flow diagram consumes data for processes as soon as the data is available. Thus, passive data flows are removed.

It is also not permitted for a node to be able to consume data from more than one input port at any given time. Allowing consumption from more than one input at a time not only increases the number of possible input and output relations of a node, but also raises the question of the meaning of ports. The idea behind a port is to define input values that need to be consumed together so that a node can use those inputs to produce output.

1.5.4 Data Types

Predicate and data types have a close relationship [72] since predicates can be used to define a new data type by allowing some values to be of a more general type, and not allowing other values of the same general type. This logical view of types, and the fact that self-referencing requires special attention in logic, will be used as the main reason not to include classes.

Classes are removed from the dialect, since they are very similar to a module and create an indirect reference between two very similar elements containing each other when an instance of a class is transported by a data flow. Not being able to use classes is a disadvantage, but it will not influence the ability to analyse the dynamic behaviour of a design.

Union types define a type that can be any one of a predefined list of types. Inclusion of union types would increase the complexity of determining the type of expression without addressing the problem statement. Therefore, union types are removed from the dialect.

The translation to SMT-LIB is only defined for numerical types and does not support the other data types allowed in the dialect. By only using numerical types, the same kind of results are expected as when using a more complex type, but without the additional computational complexity.

1.6 Associated publication

The article [4] was accepted for publication in the proceedings of the SOFL+MSVL'18 workshop, in partial fulfilment for the degree, Master of Science (Computer Science).

1.7 Outline

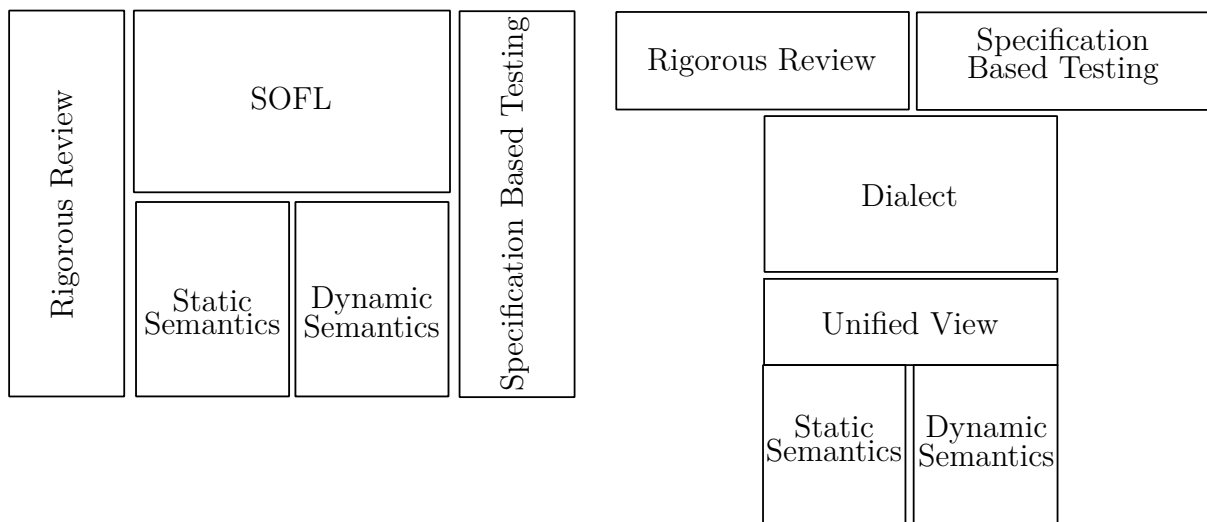
The rest of the chapters in this dissertation are divided as follows:

- Chapter 2 gives an overview of the related work that focuses on SOFL and are applicable to the work in this dissertation.
- Chapter 3 defines the dialect that is considered in this dissertation and gives a description of the semantics of the dialect by defining an action system used to describe the semantics in terms of state spaces and execution paths of a semantics.
- Chapter 4 defines a translation to SMT-LIB to define an approximation of the dialects' static semantics.
- Chapter 5 defines a translation to a process algebra to define an approximation of the dialects' dynamic semantics.
- Chapter 6 gives a short overview of the tool developed and analyses three examples in order to illustrate the ability to prove properties.
- Chapter 7 discusses future work that will add value to SOFL of the current dialect.
- Appendix A gives the grammar of the dialect.
- Appendix B gives the specification in the dialect of the example in this dissertation.

Chapter 2

Related Work

This chapter discusses previous research done on SOFL that is related to the work done in this dissertation; also, a critical discussion focusing on the semantics and verification thereof is included. Work related to this dissertation is that which focussed on the development of a semantics for SOFL, see Section 2.3. Other, less related work, is that which focussed on how to prove properties of a SOFL design, see Sections 2.1 and 2.2, but is still related since the techniques were also used to verify the static semantics.



(a) Relation of the related literature with SOFL (b) Relation of the related literature with the Dialect

Figure 2.1: Showing relation between the different areas of related work with SOFL and the dialect.

2.1 Rigorous Review

Rigorous Review is the most formal phase of the SOFL design process and is described in [46, Chap 17]. The ideas are expanded in [53], where it is called Rigorous Inspection

Method (RIM). During rigorous review, both the purpose and static semantics of a specification are verified.

2.1.1 Properties







Properties of a design that are of interest are categorised either as:

- properties that determine if a designed system is properly designed as per user requirements or
- properties of a design that are required to make the design a valid SOFL design.

Rigorous review is used to determine if any of the above types of properties are satisfied by a design.

2.1.2 Method

RTT [45] [46, Sec 17.3] is a visual notation based on fault tree that is used to prove properties based on the static semantics of SOFL. A tree is used to represent a property that needs to be proven and the property is divided into parts to create the tree. This makes it easier to prove properties at lower levels. Each node in the tree defines a sub-expression and whether the sub-expression must or can hold, by either using a rectangle or a rectangle with rounded corners, for the sub-expression to be satisfied. Below its shape is a smaller shape that defines how the children of the nodes must be interpreted; this can be either:

1. All the children's properties must hold .
2. One of the children's properties must hold .
3. All the children's properties must hold from left to right .
4. One of the children's properties must hold from left to right .
5. The property of its only child hold .
6. The property holds .

The must/can hold requirement of the expression must be read with one of the above listed properties and if the node is a leaf node, the sub-expression is assumed to be true. Figure 2.2 gives the decomposition of the predicate

$$\forall_{x \in H} (P(x) \wedge P(x + 1)) \vee (\exists_{y \in H} y \leq x)$$

using task review trees. Using the tree to evaluate the predicate, each leaf node is evaluated and the node is replaced by its value. A node with all its children evaluated in the previous step is then evaluated by applying the operation specified by the box. This process is repeated until the value of the root node is determined.

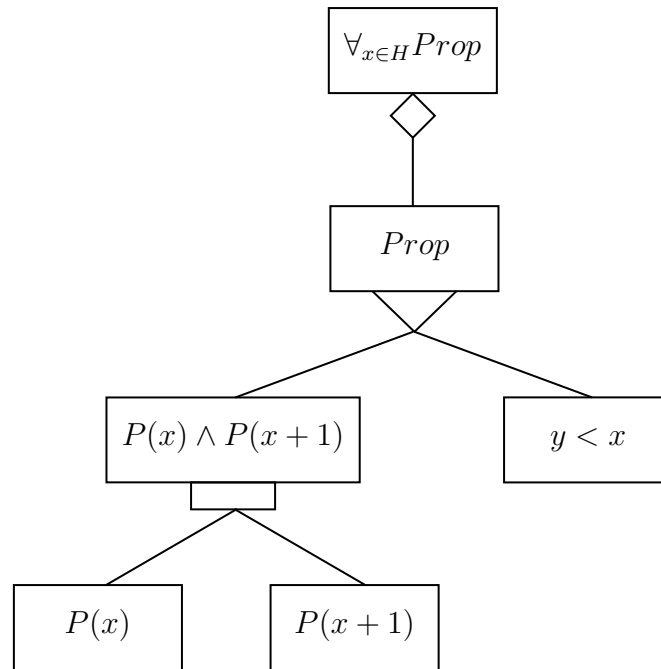


Figure 2.2: A task review tree of an expression

RIM [53] is an approach that combines FS and RTT to prove properties of a design. FSs are used to group elements from a CDFD together to describe an input/output relation that is also associated with a property that needs to be proven. Each operation involved in the FS must first be inspected after which the FS is inspected. Inspection is the process proving that a FS/operation satisfies some property using RTT.

2.1.3 Comments

RIM is a very useful approach to verify that a specification satisfies certain properties. The use of FS allows a CDFD to be further divided into groups of elements where each group is responsible for realising a specific set of properties. Since the generation of FS can be automated [55], the most labour-intensive part of RIM is the use of RTT. The use of RTT also requires the designer to have a good background in logic and a thorough understanding of the semantics of SOFL.

RTT can be used to verify the dynamic semantics¹ of a design, but as the size of the design increases, the complexity of the trees used to represent the properties also increases. This would make it unfeasible to prove certain properties of large designs.

2.2 Specification Verification

Formal Specification-Based Inspection (FSBI) is an additional inspection method based on SBT, and is introduced in [50] to support RIM when a design is verified. Using this new approach, dynamic aspects of a specification are verified by considered “execution” paths for each FS.

¹Property that relates data flows are used in [43]

2.2.1 Properties

An FS defines a transformation that transforms a state space associated with the input of the FS to the state space of its outputs. Verification of such a transform's properties is a verification of a specification's dynamic semantics.

2.2.2 Method

The method of applying FSBI requires a considerable number of steps and these are:

Derive FS. The first step is to derive functional scenarios for a CDFD, as described in the method section of [54].

Derive program paths. Create program paths. This is not an easy task [24] however a method is given in [50].

Link scenarios to paths. The program paths created need to be associated with a scenario in order to divide the analysis into smaller problems and prove the properties associated with each FS.

Analyse paths to detect defects. The paths are analysed and it is determined whether the required properties are satisfied or not.

Produce an inspection report. A report is created that reflects the findings of the specification's analysis.

2.2.3 Comments

The use of FSBI is a labour-intensive approach but does provide a “middle ground” between the formal part of the verification and its usability. The part of the analysis that is the most labour-intensive is the analysis of paths and linking them to FS. For each decision of where data flows in a CDFD are placed, an additional path needs to be created. These decisions are due to condition structures or processes that contain more than one output port.

2.3 Semantics

The semantics of SOFL defines the precise meaning of the different elements that a specification contains. This part of defining SOFL is very formal and requires the use of a logical language with well-defined concepts. Previous work done on the semantics is limited; however, those of interest are discussed here.

2.3.1 Static Semantics

Static semantics [19, 20] of SOFL is defined using the language Object-Z [21] to describe the meaning of the semantics. Object-Z is itself a formal specification that includes advanced ideas such as classes and angelic choice operation. The semantics of Object-Z

is already well-defined. These advanced concepts are very useful in defining SOFL, and a static semantics is created that defines all the needed concepts of SOFL, as required by [46].

For each of the elements in SOFL, an Objective-Z class is created to describe the properties of the element and its relation to the other elements.

2.3.2 Dynamic Semantics

Dynamic semantics [30] of SOFL was defined in order to resolve uncertainty of the semantics that existed after SOFL was introduced, [57]. Since no formal static semantics existed at that time [30], some static elements were first informally defined and are used in defining the dynamic semantics.

The dynamic semantics are defined using Structure Operation Semantics (SOS) [66] where there is separation between the “state” of the system being defined and the “rules” used to change the state of the system. Each of these rules are only applied to the “state” being transformed when the “state” satisfies some condition. Definition of the fire rule of a node is of the most interest: The node fires when the following conditions are satisfied:

1. A process p exists that can consume data.
2. The pre condition is true in the current state.
3. The post condition is true in the state after the processes fire.

When the above rules can be satisfied, the process fires and the new “state” of the system is created. Determining the next set of nodes and type of nodes that can fire is also defined with similar rules.

2.3.3 Revisit of Refinement

A fundamental investigation was done in [48] of the type of refinement relation used in formal specifications. For the usual refinement relation for a process B that is refined by a process C , the refinement is defined by requiring that the predicates

$$\begin{aligned} pre(B) &\longrightarrow pre(C) \\ pre(B) \wedge post(C) &\longrightarrow post(B) \end{aligned}$$

are true and is the same conditions required by [46, Def 22] for a refinement of process. The problem with this relation is that a process can be refined into a process that cannot be implemented. To address this problem, an extension of the refinement relation is suggested and given by

$$\begin{aligned} \forall_{s \in \Sigma} pre_A(s) &\longrightarrow \exists_{s' \in \Sigma} \cdot (port_A(s, s') \longrightarrow post_A(s, s')) \wedge \\ &(\neg post_A(s, s') \longrightarrow s = s') \end{aligned}$$

where

1. $pre_A(s)$ is the pre condition of process A and s is the state in which the predicate is evaluated.
2. $post_A(s, s')$ is the post condition of process A , s is the initial state and s' is the final state after A is applied to s .
3. Σ denotes the set of all states.

This additional relation ensures that if the post condition of the process cannot be satisfied in a new state, then the new state and the initial state are equal. Thus, the complete refinement relation (with simplifications) is defined by

$$\begin{aligned}
 pre(B) &\longrightarrow pre(C) \\
 pre(B) \wedge post(C) &\longrightarrow post(B) \\
 \forall_{s \in \Sigma} pre_C(s) &\longrightarrow \exists_{s' \in \Sigma} \cdot (post_C(s, s') \vee s = s').
 \end{aligned}$$

The right-hand side of the last formula states that there exists a state where the post condition is true or the state remains unchanged.

2.3.4 Comments

The semantics of SOFL was defined after the method was created, which is not unusual, but the dynamic semantics was formalised in 1997, whereas the static semantics was only formalised in 1999 and 2002. This creates a curious situation where the dynamic semantics is based on only a semi-formal idea of the static semantics. By not having a formalised static semantics, it is not always clear that the static semantics will allow a model, therefore the dynamic semantics can be based on a model that does not exist.

Object-Z was used [19, 20] to define a static semantics using advance concepts of Object-Z. Advance concepts are useful and assist considerably in defining the semantics; however, the author does not agree with their use. Object-Z is a complete specification as well as VDM on which predicates of SOFL are based. By using a language with almost the same “level” of abstraction to define the semantics of another language, uncertainty is created, as some concepts might not be well-defined. In [20] the semantics of a SOFL class is defined by an Objective-Z class. For this approach to be acceptable, SOFL is required to be defined by a fragment of Object-Z, and therefore the semantics of SOFL is automatically defined by Object-Z’s semantics. This approach is very similar to defining a programming language² in terms of another language. For a programming language, this approach is acceptable, since the original defines the semantics and the expressibility of the new language.

Dynamic semantics defined in [30] does not follow a constructive approach. The approach follows defined rules and only models that satisfy those rules are allowed. This is a proper approach to define a semantic, but it sometimes lacks an intuitive³ understanding of the semantics. Conditions for the fire rule require knowledge of the “future” when the post condition of a node is used to determine whether a node is permitted to fire. The

²For example: Xtend, <https://www.eclipse.org/xtend/documentation/index.html>

³This is partially addressed in [30] by using comments for the rules defined.

properties of non-existing output are required to determine if output should be generated: this is a curious situation since a data flow diagram processes data from its input to output without first testing the output that could be created. A node in a data flow diagram does use knowledge of availability of data on its output data flows to determine if the node can fire. The information that the node uses is not “future” knowledge as the node only fire if there is no token on output data flows, and only the node can generate on the output data flows.

The update refinement relation of [48] is not applicable to SOFL, as state changes are only influenced by nodes when they fire. There is no reason for pre and post condition to be used as a “double” check to make sure the fire rules function correctly. SOFL defines only one post condition for all the output port of a node where any of the ports are allowed to generated data. This could allow unwanted scenarios during refinement as described in [48].

2.4 Summary

Both Rigorous Review and Specification Base Testing are more suitable to verifying properties that are required by the system and not to verify semantics, since properties of a system are more diverse and those required by the semantics are fixed when the language is defined.

Both static and dynamic semantics are very old and are seen as too abstract for a formal specification language. Better separation between pre and post condition and the firing rules of nodes is needed.

Chapter 3

Description of SOFL Dialect

The SOFL dialect specifications are defined using text files, but for clarity of this dissertation the graphical representation for CDFDs is still used. Section 3.1 gives an overview of how the dialect specification is defined with particular focus on the syntax changes made to create the dialect. For a complete description of the grammar see Appendix A; familiarity with [46] is also assumed.

A semi-formal description is defined in Section 3.2 and is used as a starting point when static and dynamic semantics are defined.

3.1 Syntax

In this section, examples are given for each component to illustrate how the syntax of the dialect is different from those of original SOFL.

3.1.1 Module

A module is a container used to group the elements in a design. When the module inherits from a parent module, the module also inherits the variable and types in the parent module. Processes are defined in a module, and they define the computational units in a specification. Functions are only used as part of an expression. The behaviour of a module defines how processes in a module are used to realise a module's behaviour. An example of a module's specification is given below with comments:

```
module NameOfDesign / ParentModuleName
/*Type declaration*/
type
  TypeName = Type;
  AnotherTypeName = AnotherType
/*
Variable declaration
*/
var
  ext # wd external_to_system : Type;
```

```

    ext rd store_of_parent;
    new_store : Type
  /*
  Constant declaration
  */
  const
    constant_name = /*expression*/;
    another_constant = /*another_expression */
  /*Invariant declaration*/
  inv
    Invariant_one;
    Invariant_two
  behaviour /*Behaviour specification*/
  end_behaviour
  /*Process specification*/
  /*Function specification*/
  endmodule

```

Only predefined types, as defined in [46], can be used in a module to define new types, and defined types can be used in all the modules that inherit from it directly or indirectly. If a module reuses a type name, then the new type “over shadows” the previously defined type.

The variables are either defined in the current module or accessed from an external source. Variables from an external source are indicated by `ext #` and can have either read or write access. Other external variables are inherited from the parent module after the stores are filtered by the process that this module refines. These are indicated by the keywords `ext rd` or `ext wr`, depending if read or write access is required to the variable. Constants are special variables, as after their initialisation by an expression they have read-only access.

Invariants are used to define predicates that must be evaluated to true between times when processes fire, and invariants can only access variables that are visible in the current module.

The behaviour, processes, expressions and functions are considered below since they are more complex than the other elements.

3.1.2 Behaviour

Behaviour elements are textual representations of a CDFD and define how a process is being refined. They have input and output ports related to the process being refined. An example of a graphical representation is given in Figure 3.1 and a textual representation is given by:

```

behaviour BehaviourName(
  tmp_data:real | middle: real | tmp_added: real)
  out_data:real | out_added: real

```

```

/*
Define addition input/output data with their data types
*/
added(added_input:real) added_output:nat0
/*
When addition flows are added the input/output ports
need to be redefined. Type information is not added
since they are redundant.
*/
grouping(tmp_data | middle,added_input | middle | tmp_added)
      out_data | out_added, added_output

state_condition in 1 out 1 pre true post true
...
state_condition in 4 out 2 pre true post true
/*
Structure node definitions
*/
/*
Data flow definitions
*/
end_behaviour

```

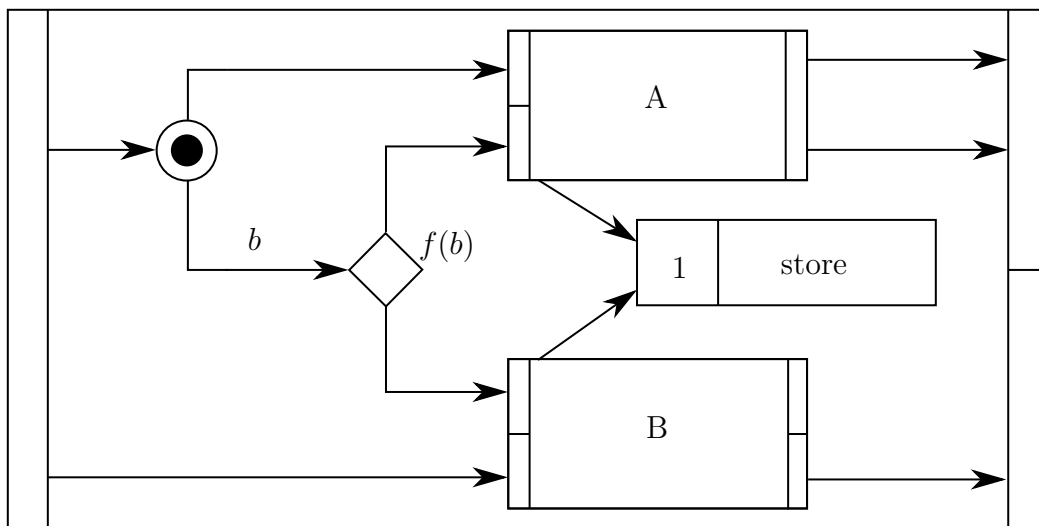


Figure 3.1: Example of a CDFD. The CDFD is drawn inside a process symbol since it must also contain input and output ports in the dialect

The meaning of a CDFD is based on a data flow diagram [17], where each computational element processes data as soon as all its input data flows contain data. In the case of CDFDs, data flows are grouped to indicate from which input flows data can be consumed, and on which output flows data can be generated: these groupings are called ports.

The behaviour element is used to refine a process, and the process being refined determines restrictions on the input and output behaviour of the CDFD. In order for the CDFD to perform the transformation from input to output, it might be necessary for additional input or output data flows to be defined. These additional data flows and the data flows from the process being refined also need to be grouped by ports, for the input and output of the behaviour element to be well-defined. Newly introduced data flows must be associated with at least one port, but may be associated with more. This is allowed since the process being refined will determine when the CDFD fires.

A CDFD defines how input are transformed into output and therefore also define a predicate transform. The pre and post conditions are made explicit for each pair of ports using a `state_condition` statement block. The predicate does not define a state change in the CDFD only the allowed input and output states, whereas the data flows and nodes are used to realise the state change.

Control nodes assist data flows to transport data to the process that will perform the needed computations. The permitted control nodes are:

1. Conditional structures
2. Merging and separating structures
3. Broadcast structures

The remainder of this section describes these components and gives a graphical representation of data flows, data stores and the control nodes.

Data flows

Data flows are the carriers of data between different components in a CDFD. Each data flow is associated with a fixed type and is in one of two states: having data or not having data.

Two types of data flows are used, these are: active and shadow flows, and are defined by indicating the node and variable connected to the flows where the left-hand side of `=>` is the source and the right-hand side is the destination.

```
flow behavior.tmp_added => proc Combine.tmp_added
flow proc Combine.out_added => behavior.out_added
shadow flow cast Broad.out_added => proc Combine.out_added
```

An active flow is defined by `flow` and a shadow flow by `shadow flow`. After the node identification the variable name is given, except for the behaviour which does not require a name, as only one such element is contained in a module. The source and destination name must be the same as well as their type. In the dialect, it is not possible for a port not to be connected to a data flow, and shadow flows are introduced for this purpose. The only difference between shadow and active flows is that the value carried by shadow flows meaning is limited to the presence or absence of data: the fire rules are the same.

Conditional structures

Two representations are given for a conditional structure in Figure 3.2. When only two output branches exist a diamond shape is used. When more than two output branches

exist, the pairwise conjunction of the predicates must be false and disjunction of all output predicates must be true, given the allowed input values. A textual specification is given by:

```
condition ConditionName(input)
```

```
  out1: out1 > 0,
```

```
  out2: out2 < 0,
```

```
  out3: out3 = 0
```

```
end_condition
```

where the type of the output variables are the same and are defined by the input variable. These variables are connected via data flows to other variables contained in other elements.

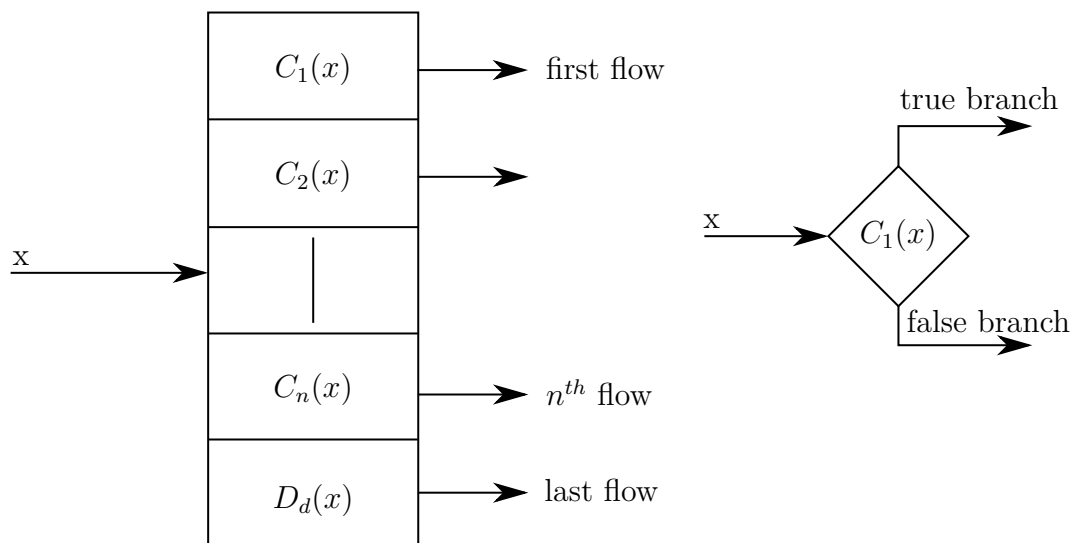


Figure 3.2: A conditional structure representation.

Merging and Separating structures

A merging structure takes n input values and produces one value with a type equal to the product [46, Chapter 10] of the input types, and each component of the output is equal to their respective input value. Separating structures undo the effect of merging structures (see Figure 3.3). The purpose of these structures is two fold:

1. To reduce the number of data flows in a diagram
2. When the condition of a condition structure depends on more than one data flow, a merging structure creates one data flow required by a conditional structure.

```
merge MergedName(var1, var2, var3)
```

```
  out_combined
```

```
end_merge
```



```

unmerge UnmergedName(in_combined)
  var1,
  var2,
  var3
end_unmerge

```

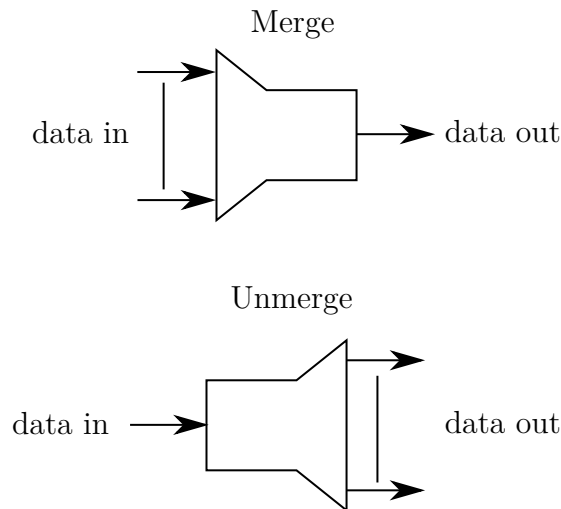


Figure 3.3: Merging and separating structure representation

Broadcast structures

A broadcast structure that duplicates the input and places it on all output flows without changing the data value, is shown in Figure 3.4. The textual specification is given by:

```

broadcast Cast(indata)
  out_data_1,
  out_data_2,
  out_data_3
end_broadcast

```

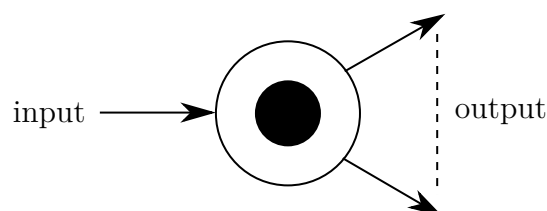


Figure 3.4: A broadcast structure representation

Data stores

A variable defined in the module is also referred to as a data store when used in a CDFD. Figure 3.5 shows how data stores are associated with a process. Process *A* has write

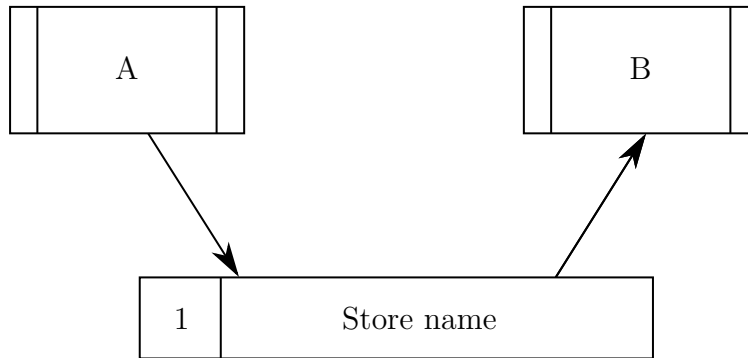


Figure 3.5: A graphical representation of a data store

access whereas process *B* has read access to the data store. The type of the data store cannot change during the lifetime of the data store.

3.1.3 Processes

Processes are the elements in a module that perform computations as well as providing the opportunity to refine process by another module. An input/output state transform is defined similarly to a behaviour element, and the pre/post conditions are defined by `state_condition` statements. Process also defines access required to data store that are visible in the current module, and only those data store which a process has access to can be inherited by the module that refine the process.

A graphical representation of a process is given in Figure 3.6; its textual specification is given by:

```

process NameOfProcess(sel: ServiceCollection | current_inf: CustomerInf)
    current_inf1: CustomerInf |
    current_inf2: CustomerInf |
    e_mesg1: string

    /*reference variables visible in the containing module*/
    ext wr ss3
    state_condition in 1 out 1 pre out1 > 10 post loop_d = ~out1
    ...
    state_condition in 2 out 2
        pre loop > 0 post loop = CallToFunction(~loop)
end_process
  
```

In the last predicate transform defined, the function `CallToFunction` is applied to the variable `loop` as it was before the execution of a process, by referring to it by `~loop`. The pre/post condition of the process does not define a change of state stores or data flows values; they only define the state in which the process is allowed to execute and the allowed state after execution. State change is realised by defining a module that refines the process.

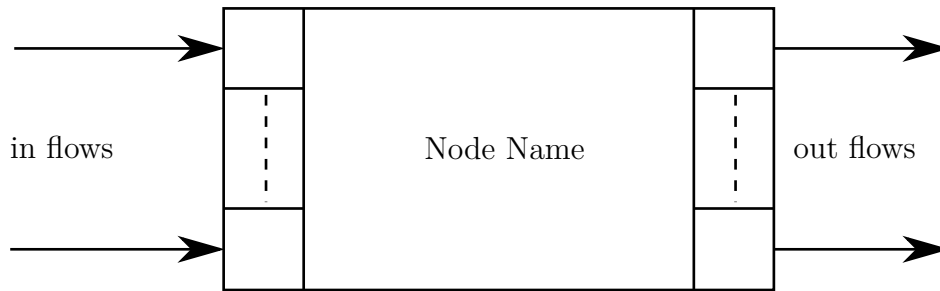


Figure 3.6: A graphical representation of a process

3.1.4 Expression and Function

Expressions form a large part of a specification as they are used to define predicates, constant values, and the initial state space of a CDFD. Expressions are defined in [46], but it is important to note that an expression cannot change an input value and cannot use a process in its evaluation. Functions are used as an abbreviation for an expression and can only be used as part of an expression. An example of a specification is given by:

```

function FunctionName(param1:Type1, param2:Type2) : ReturnType
  pre PreconditionExpression
  post PostConditionExpression
  = Expression
endfunction

```

3.1.5 Syntax Differences

The grammar of SOFL [46, Appendix A] is modified to obtain a grammar for the dialect and the exact locations where the grammar was modified are indicated in the Appendix. This section gives the changes made to the grammar of SOFL. Whenever a syntax rule is removed or modified which results in other rules no longer being referenced in the grammar, those rules are also removed from the grammar. The modified rules are not named explicitly, but rather the keyword that is removed is used.

CDFD and Module Related

Removal of the visual representation of a CDFD has considerable effect as the “name” of the CDFD cannot be used in the specification. The visual representation is replaced by a `behaviour ... end_behaviour` statement block that contains all the definition of elements of interest that were defined in the visual representation of a CDFD, and is used to define the CDFD in the dialect. The statement block also replaces `behav` which is used in SOFL to associate a CDFD with a module, but in the dialect the CDFD is part of the module.

SOFL is allowed to reference previously defined elements in other places of the specification. Syntactical element of the form `Identifier('.'Identifier)*` and references of previously defined elements are removed.

Process Related

Definition of a process's pre and post conditions is replaced by a `state_condition` statement for each input and output port pair.

When a process is refined, the keyword `decomposition` is used in the dialect and uses the name of the module that refines the process. SOFL use the keyword `decom` and the name of the CDFD that refines the process.

The initial state of a module in the dialect is defined using predicates and the `Init` process, whereas SOFL uses an explicit specification. An example of a `Init` process in the dialect is given by:

```
process Init()  
  ss3 < 20;  
  ss3 > 11  
end_process
```

Explicit specification of processes is not allowed, thus the keyword `explicit` is removed.

Types related

The types removed are “classes”, “undefined” (part of LPF) and “universal” as well as keywords used to define element and functions defined in SOFL that use removed types as parameters. Thus, the removed keywords are:

1. `universal`
2. `sign`
3. `nill`
4. `undefined`
5. `bound`

Expression related

The most notable change to expression is that no “imperative” expression is used, thus assignments are only allowed as part of a `let ... in ...` statement block. Also, all elements of expression that are used with types removed are also removed.

SOFL also allows a quantifier over a collection of data values, e.g.

```
forall[e1: list containing values | expression]
```

where the dialect only allows use of data types over which a quantifier can be used.

3.2 Semi-Formal Semantics

This section starts by defining the semantics using an intuitive view of how a program can be represented by state graphs (program graphs), Section 3.2.1. This is followed by how the predicate transform represents the state change by a node in these graphs

(Section 3.2.2). Predicate transforms are then used to describe the input and output state spaces of a CDFD. From these input / output relations an action system (Section 3.2.4) is created that gives a formal description of a CDFD's program graph.

The hierarchical structure created by a CDFD refining processes (Section 3.2.3) is used to combine the semantics of different CDFDs together. No dynamic behaviour is defined between levels in the hierarchical structure.

Throughout this process, conditions are defined that need to be satisfied for the final semantics to be sensible. This is done to place additional focus on those properties and motivates how those properties are applicable to the dialect. Finally, the conditions are also used to determine which properties can be verified during either static or dynamic verification.

3.2.1 Program Graph

A program graph¹ is used to present how different elements of a program execute; where each vertex in the graph defines the state of the program. A transition from one state to another is labelled by all the nodes that fire in a CDFD at a specific time. This view is different from a data flow diagram of [17] in which the operations are represented by vertices. The two views are equivalent but the meaning of vertices and edges is swapped.

A program graph is given in Figure 3.7 of the function:

```
function func(x)
  if x > 0
    return x + 1
  else
    return 2*x
```

Each of the paths in the graph give a sequence of state transitions that are used to compute the results of the program. Such a path is called an execution path and this path is used to represent the sequence of computation performed. By allowing a program to perform more than one operation per transition in the execution path, parallelism is represented, therefore such parallelism is inherently part of a CDFD since it is based on data flow diagrams.

For a CDFD, a program graph is created using Algorithm 1. To be able to construct a program graph, the fire rules of a node are used and are dependent on the data flows connected to the node's input and output ports:

1. There is only one input port of the node such that all the data flows connected to that input port contain data.
2. There does not exist a data flow connected to any output port of the node that contains data.

An execution path of the program graph starts with a state in *Init* and terminates in a vertex with no outgoing edges, and a path is allowed to contain only one state contained

¹The possibility to derive FS by using a topological (e.g., program graph) structure of a specification is stated in [38].

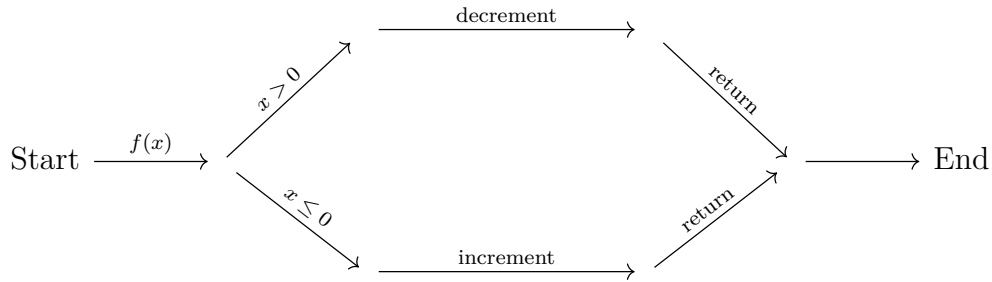


Figure 3.7: Example of program diagram

Data: A CDFD

Result: A program graph $(Init, Term, N, E)$, the set of initial vertices, terminating vertices, all vertices and edges, respectively.

Each input port of the CDFD define an initial state space, these vertices are put into the sets $Init$ and N ;

Put all the vertices in $Init$ onto the stack S ;

for *pop a vertex v from the stack S* **do**

 Let I be the set of nodes that can fire in state v ;

 Let C be a collection consisting of a set of nodes, where each set is a duplicate of I but marked with a different combination of input port (to consume data) and an output port (generate data);

for $c \in C$ **do**

 Let e be the state when the set of nodes c transforms the state v , i.e., the needed data is consumed from v and the nodes generate generate data to create the state e ;

 Add e to N and the transition $v \xrightarrow{c} e$ to the set of edges (E) ;

end

end

for $n \in N$ **do**

 If the state n generated data on an output port of the CDFD, add n to the set $Term$;

end

Algorithm 1: Generation of a program graph from a CDFD

in *Term*. It is also required that all execution paths must be of finite length so that a CDFD terminates; no infinite loops are permitted.

The dialect also allows for refinement of process by a CDFD that is not addressed by a program graph. Both refinement and realisation of “execution” paths will use predicate transforms.

Conditions to Satisfy

Only execution paths that start with a state space defined by an input port of a CDFD and generate data on only one output port, are allowed.

Condition 1. Each execution path of a program graph starts with a vertex in *Init*.

Condition 2. Each execution path must contain only one vertex in *Term*.

Data flow diagrams are generally used to describe “functions” which need to produce output, and the same ideas are applied in the dialect. An additional condition for a CDFD to be able to generate output is that it must terminate. Termination is ensured by only allowing designs where all “execution” paths are of finite length.

Condition 3. Each execution path of a program graph must have finite length.

3.2.2 Predicate Transforms

Predicate transforms are used to represent how the current state is transformed into a new state by a node. A predicate transform is created for each node as well as for each execution by combining the state space transformations of each set of nodes.

State Space

A state space is a collection of sets, where a set consists of variables associated with allowed values for data flows and data stores. When a state space is considered and no distinction is needed between data flows and data stores, they will be referred to as variables. The state space σ is identified by a predicate $predicate(\sigma)$, where the predicate evaluate to true only for the element in the state space. When a variable is not used in $predicate(\sigma)$, the predicate does not restrict that variable. Note: σ is the largest possible state space for which $predicate(\sigma)$ will evaluate to true, i.e., $predicate(\sigma)(a)$ is true for all $a \in \sigma$ and false if $a \notin \sigma$. A state space ς is contained in a state space σ ($\varsigma \subset \sigma$) if:

1. $a \in \varsigma \cap \sigma$; $predicate(\sigma)(a)$ and $predicate(\varsigma)(a)$ is true, and
2. $a \in \sigma \setminus \varsigma$; $predicate(\sigma)(a)$ is true and $predicate(\varsigma)(a)$ is false, and
3. $a \in \bar{\sigma}$; $predicate(\sigma)(a)$ and $predicate(\varsigma)(a)$ is false.

Thus it is the case that $\varsigma \subset \sigma$ if and only if $\forall (predicate(\varsigma) \longrightarrow predicate(\sigma))$, where the quantifier is only over all the free variables used in $predicate(\varsigma)$ and the formula is abbreviated by

$$predicate(\varsigma) \longrightarrow_o predicate(\sigma) \tag{3.1}$$

when the addition condition $\varsigma \neq \emptyset$ is also included.

Weakest Precondition

A node is used to transform values in a CDFD, and this transformation is defined using pre and post conditions. A predicate cannot transform a state variable associated with the state space before the node fires, therefore state changes are indicated in the post condition by accessing input data of a node by adding a prefix to the name of the data. For an input variable \mathbf{x} , the output state use $\sim\mathbf{x}$ to refer to the original input values.

The weakest pre condition [18] will be used as a predicate transform

$$\mathbf{wp} : A \times \Sigma \rightarrow \Sigma.$$

with A the set of programs being described.² The initial and final state of processes are described by predicates.

For a program a and a state space σ consisting of all valid states after execution, the weakest pre condition $\omega = \mathbf{wp}(a, \sigma_f)$ is the largest possible state space in which the program a can execute and result in a state contained in σ . These state spaces are not always exactly defined in a specification since predicates are used to define the state spaces and do not necessarily define the state space exactly; i.e., the predicate can change during refinement. To handle this uncertainty, use of state spaces denoted by Greek letters (e.g., σ) will denote the “ideal” state space as intended, where the pre or post conditions are used to denote approximation to the state spaces.

Therefore, for the pre and post condition of a program a to be correctly defined it must be the case that: For a pre condition $pre(a)$ of the program it must be the case that

$$pre(a) \longrightarrow_o predicate(\omega)$$

and for its post condition $post(a)$ it must be the case that

$$post(a) \longrightarrow_o predicate(\sigma)$$

as well as

$$pre(a) \longrightarrow_o predicate(\mathbf{wp}(a, post(a))). \quad (3.2)$$

and is illustrated in Figure 3.8.

“Execution” Paths

“Execution” paths are used to describe the state transform from an input state space of a CDFD to its output state space. For each transition in an “execution” path a set of nodes A_i fires. Consider an execution path

$$\cdots \sigma_{i-1} \xrightarrow{A_i} \sigma_i \xrightarrow{A_{i+1}} \sigma_{i+1} \cdots$$

with σ_i the state spaces and A_i the set of nodes that result in state transition, which also define a predicate transform such that $\sigma_i \subset \mathbf{wp}(A_i, \sigma_{i+1})$. For the “execution” path to be

²The notation $\mathbf{wp}_s(x)$ will sometimes be used to denote $\mathbf{wp}(s, x)$.

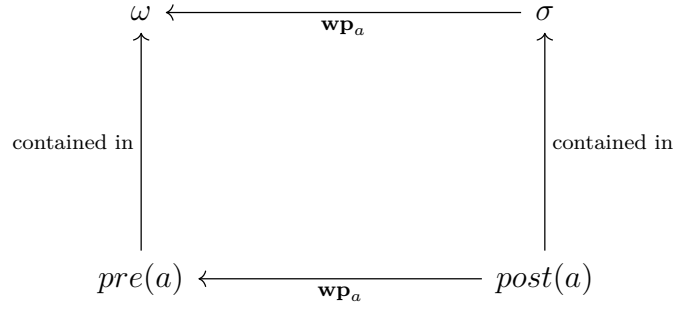


Figure 3.8: Relationship between pre/post condition and the weakest pre condition

valid, it must be the case that for any two consecutive predicate transforms:

$$\begin{aligned}
 \omega_{i+1} &= \mathbf{wp}(A_i, \sigma_{i+1}) \\
 \sigma_i &\subset \omega_{i+1} \\
 \omega_i &= \mathbf{wp}(A_i, \sigma_i) \\
 \sigma_{i-1} &\subset \omega_i
 \end{aligned}$$

and

$$\begin{aligned}
 post(A_{i+1}) &\longrightarrow predicate(\sigma_{i+1}) \\
 pre(A_{i+1}) &\longrightarrow predicate(\mathbf{wp}(A_{i+1}, post(A_{i+1}))) \\
 pre(A_{i+1}) &\longrightarrow predicate(\omega_{i+1})
 \end{aligned}$$

and is illustrated in Figure 3.9.

Two consecutive predicate transforms can only be combined if it is the case that

$$\sigma_i \subset \mathbf{wp}(A_i, \mathbf{wp}(A_{i+1}, \sigma_{i+1}))$$

and all relation in Figure 3.9 are valid. This concatenation is illustrated in Figure 3.9. By repeating this process a predicate transform is created from an execution path, and must match an input/output pair of the CDFD being analysed.

By repeating this process, and since all traces are finite, a predicate transform for the execution path is created, given that no state space σ_i is empty. A predicate transform created from an execution path must match an input/output pair of the CDFD being analysed.

Invariants of a module are defined by predicates that must be true before and after a set of actions fire. Thus, for a state space σ_i and invariant inv , it must be the case that an assignment must exist that makes the formula

$$predicate(\sigma_i) \longrightarrow_o inv$$

true, i.e., satisfiable. The state in which an action system (CDFD) is initialised is defined by the predicates defined in the process **Init**, and the conjunction of those predicates

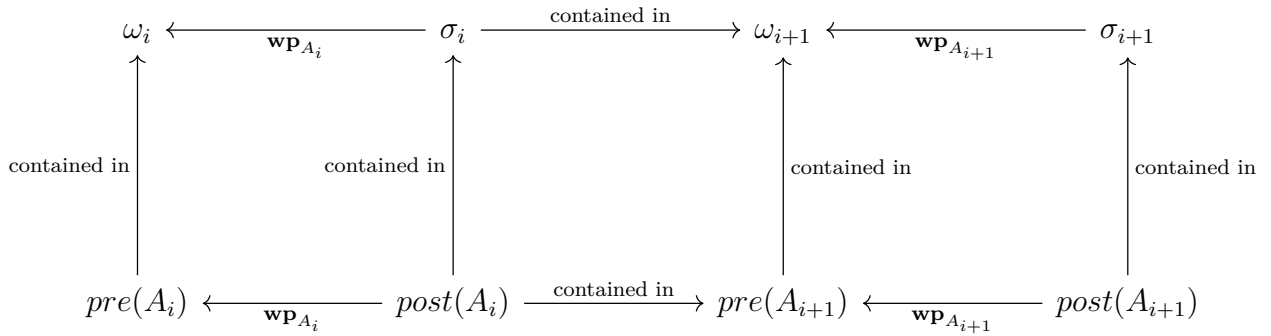


Figure 3.9: Relationship between pre/post condition and the weakest pre condition and “execution” paths

gives *init*. For the initialisation to be valid it must be the case that

$$init \longrightarrow_o inv$$

otherwise the invariant is immediately false upon initialisation.

Conditions to Satisfy

Throughout an “execution” path, no empty state spaces are allowed, since an empty state space means that assignment to variables is not allowed. The designer should have the option of not restricting all variables by predicates which would mean the allowed variables are not being tested.

Condition 4. No empty state spaces are allowed in an execution path.

All state spaces “connected” to an input port of a CDFD must allow the initialisation condition to be true. This creates a data flows dependency by creating state “freeze” until all nodes in the current execution path, that are connected to the input port, fire.

Condition 5. The initialisation conditions must be valid in the first state space of an execution path.

Invariants define a “global” restriction on the allowed state space of a CDFD, and it is well-defined³ when the invariant is true.

Condition 6. The invariant must be satisfiable in each state space of an execution path.

The only⁴ persistent state elements are data stores which can be accessed concurrently and therefore read/write mutual exclusion is a requirement for their state to be meaningful. The only time when their state can be corrupted is when a set of nodes fires, and those set of nodes will be tested to determine if there are any violations.

³This is also true in SOFL

⁴Data flows also define state of data flow diagram but are managed by the fire rules of a node.

Condition 7. Each set of actions in Act must access data stores so that there are no read/write violations.

An “execution” path is created from predicate transform which, in turn, define a transformation between input and output state spaces.

Condition 8. Each execution path consists of a sequence of predicate transforms that can be combined with the predicate transform of the execution path.

In the dialect all ports must be connected to a data flows; this is ensured by the syntax of the dialect. It is also required that all nodes are connected by data flows. If a node does not appear in an “execution” path it is an indication that further investigation is needed to determine why the node does not fire.

Condition 9. All port need to be connected to another port. When active data flows are used the same variable name must be used in each port. When shadow flows are used a “dummy” variable is created, and the names for connected port must also be the same.

Predicate Transform of a Node

A node consists of a number of input and output ports, where each pair of input and output ports define a predicate transform. Thus, set A_i of node must also indicate which input and output port of each node are involved in transforming the state spaces. The state spaces ω_i and σ_i (in Figure 3.9) are composed of the input/output state spaces of each node in A_i , and the decomposition will be based on predicates of the node’s input and output port pairs.

For a node n with input ports I and output ports O , the pre condition of port ($i \in I$) is defined by

$$pre(n, i) = \bigvee_{o' \in O} pre(n, i, o')$$

where $pre(n, i, o)$ is the pre condition where input port i consumes data and output port o generates output. Similarly, the post condition of an output port ($o \in O$) is defined by

$$post(n, o) = \bigvee_{i' \in I} post(n, i', o)$$

where $post(n, i, o)$ is the post condition where input port i consumes data and output port o generates output. The pre condition for the set A_i is defined by

$$pre(A_i) = \bigwedge_{a \in A_i} pre(a, i_a, o_a)$$

where i_a and o_a are respectively the input and output ports of a node a that consumes and generates data, similarly the port condition is defined by

$$post(A_i) = \bigwedge_{a \in A_i} post(a, i_a, o_a).$$

The pre condition $pre(n, i)$ and post condition $post(n, o)$ of a node n are used during refinement, whereas the pre condition $pre(n, i, o)$ and post condition $post(n, i, o)$ are used when an “execution” path is created. Thus, two types of predicate transform are associated with a node:

$$pre(n, i) \rightsquigarrow post(n, o), \text{ or}$$

$$pre(n, i, o) \rightsquigarrow post(n, i, o).$$

During refinement the predicate transform $pre(n, i) \rightsquigarrow post(n, o)$ is refined by a CDFD and is used to motivate that the node can be implemented, see Section 3.2.3. When an “execution” path is considered, the predicate transform $pre(n, i, o) \rightsquigarrow post(n, i, o)$ that is dependant on the specific input and output ports and the information of the refinement is not available, therefore it is only required that there is no contradiction between the pre and post conditions of a node.

A node is assumed to be contradiction free if

$$pre(n, i, o) \longrightarrow_o post(n, i, o).$$

is valid for input i and output o ports of n .

Conditions to Satisfy

Not all nodes can be defined by using only their pre and post conditions and neither can the post conditions of a node always be proven from its pre condition. This is usually the case for node at a high level in the refinement hierarchical structure. It is only required that node does not define any contradiction⁵ between its input and output ports.

Condition 10. The process being refined must be realisable, i.e., there is no contradiction between its predicates.

It is possible for a node to consume data from more than one input port at a time. A dependency is created between sets of nodes that fire which is seen as against the “spirit” of SOFL to be an intuitive design method. Consider a node that can consume from more than one input port. The node will consume data from one input port and generate data on its output flows which will prevent the node from firing immediately again. For the fire rules to be implemented correctly they need to predict the “future” which is possible in this situation, but the author decided against it as it would require similar properties needed for a non-deterministic structure to be included in the dialect.

Condition 11. A node is not allowed to consume data from more than one input port at any given time.

3.2.3 Refinement

Refinement occurs when addition information is added to a specification as part of the design process, see Figure 3.10 that shows a process (top) and a CDFD (bottom) that is

⁵A contradiction will result in an empty state space.

used to refine the process. Additional input ports are defined for the CDFD (indicated by green) due to the fact that a CDFD can define new input and output data flows. The part of the refinement that handles the requirement that the CDFD must realise the functionality of the process, is indicated by the upward and downwards blue arrows in the figure.

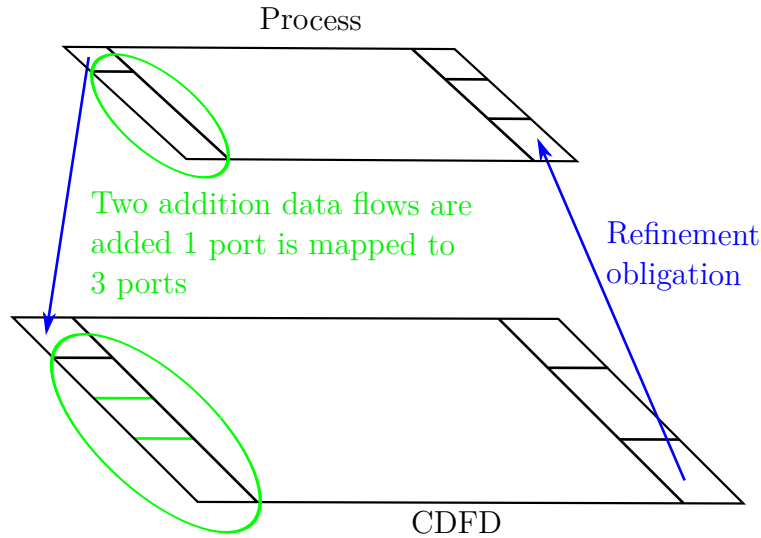


Figure 3.10: Representation of a process being refined: the green element indicates where ports are added to a CDFD, and the blue element, the refinement part, that works with predicate transforms.

The predicate transform of the process is given by \mathbf{wp}_{S_n} and for the CDFD by \mathbf{wp}_{S_c} , where each predicate transform operate on “different” state spaces and the refinement links these state spaces. During refinement Figure 3.9 changes to Figure 3.11, where A_i is the set of nodes that fire in the CDFD to realise an input/output port combination of the process. Containment of the state space ω also changes direction since ω defines the state space in which the CDFD fires.

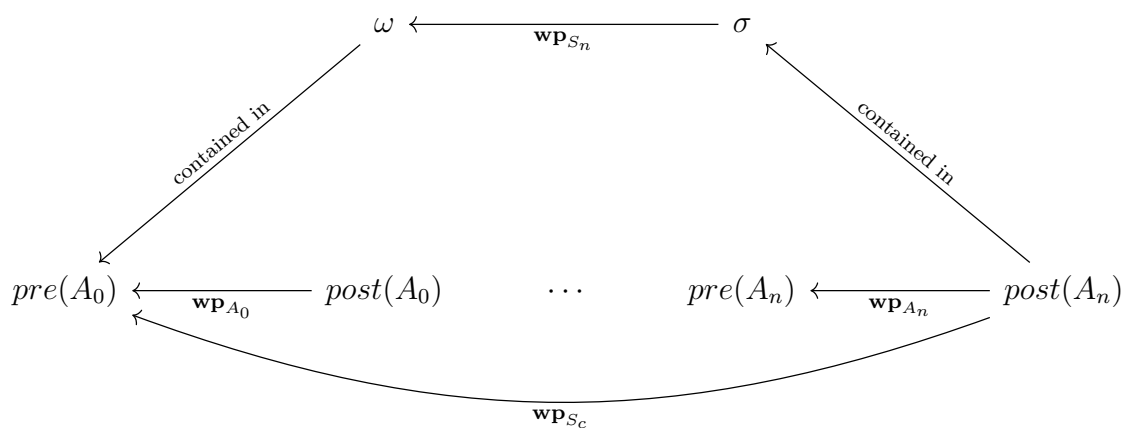


Figure 3.11: Relationship between pre/post condition and the weakest pre condition, during refinement

The difference between the state space of the process and the initial and final state space of the CDFD is that the CDFD can have additional data stores and data flows. For the same data store and data flows in both the process and CDFD the same variable name is used, and therefore containment is tested by:

$$\begin{aligned} \text{predicate}(\omega) &\longrightarrow_o \text{pre}(A_0), \text{ and} \\ \text{post}(A_n) &\longrightarrow_o \text{predicate}(\sigma). \end{aligned}$$

The state spaces ω and σ are not exactly defined by the predicated but it will be assumed that the state spaces are exactly defined for the discussion. Therefore,

$$\begin{aligned} \text{pre}(n, i) &= \text{predicate}(\omega) \text{ and} \\ \text{post}(n, o) &= \text{predicate}(\sigma) \end{aligned}$$

for the input port i and output port o that define the transition of the process being refined.

For $\text{pre}(A_0)$ and $\text{post}(A_n)$ the exact state spaces are not defined by the CDFD's pre and post conditions. Each input and output port pair of a CDFD is connected by a number of "execution" paths. Let ϕ_k be the initial state spaces and ς_k the final state spaces of the k^{th} "execution" path that generates output for the CDFD. Also, port of the process are associated with ports of the CDFD by using a mapping \mathcal{G} . The mapping \mathcal{G} maps a port identifier of the process to a set of port identifier of the CDFD. Note: newly introduced data flows can only be associated with one port of a process. Thus, for the purpose of refinement it will be assumed that

$$\begin{aligned} \bigvee_{j \in \mathcal{G}(i)} \text{pre}(c, j) &= \bigvee_k \text{predicate}(\phi_k) \text{ and} \\ \bigvee_{j \in \mathcal{G}(o)} \text{post}(c, j) &= \bigvee_k \text{predicate}(\varsigma_k). \end{aligned}$$

Thus, for a process p with input port i and output port o , refinement by a CDFD c is valid if the predicates

$$\begin{aligned} \text{pre}(p, i) &\longrightarrow_o \bigvee_{j \in \mathcal{G}(i)} \text{pre}(c, j) \\ \bigvee_{j \in \mathcal{G}(o)} \text{post}(c, j) &\longrightarrow_o \text{post}(p, o) \end{aligned}$$

are valid for each port i and o .

Condition to satisfy

All external data stores accessed by a CDFD's state space need to be restrictable for them to be usable in the initialisation conditions, in the dialect. The mechanism allowed in SOFL to enforce such restrictions is that the process being refined has access to the data store.

Condition 12. All external data store accesses by a CDFD must be accessible by the process the CDFD refines.

Input and output data flows for a CDFD need to be properly defined and grouped by an input or output port of the CDFD.

Condition 13. Whenever a CDFD introduces new input (or output) data flows a new input (or output) port can be created, and added to input (or output) ports of the CDFD.

A CDFD needs to behave like a node (e.g., terminate) before it will be sensible to use the CDFD as part of a refinement process.

Condition 14. The data flows analysis of the CDFD involved in refinement must be valid before the refinement process can be considered as valid, e.g., each execution path must be finite.

The refinement step in “Correctness by Construction” method is performed in the dialect by verifying that each input port realise a functionality described by the data flow diagram. The condition below is very strict, requiring that a path exist for each input and output port combination. In this dissertation relaxation of the condition was not considered; it was left as is.

Condition 15. A CDFD must realise at least one execution path for each input port of the process being refined and for each output port of the CDFD.

3.2.4 Action Systems

Here a program graph and predicate transforms are combined and extended to describe the semantics of a CDFD. The result is called an action system (Definition 1) and is based on ideas used in [1] where a different action system is defined.⁶ A combination of dynamic and static semantics is created by the action system, and later separated when the static semantics (Chapter 4) and the dynamic semantics are discussed (Chapter 5).

Definition 1. An action system is a tuple

$$ActSys = (Act, Var, Init, Inv, Actions, Term, N, E)$$

where the components are defined as:

1. *Var*, is a set of elements where data is stored.
 - (a) Data store: is a persistent storage element where nodes access the data store with either read or write access.
 - (b) Data flows: is a persistent storage element which is only accessible by its two connected ports, where the one port generates data on the data flows and the other consumes data from the data flows. Only when data is generated on a data flows can the data be consumed, and only once per generation.

⁶Here the fire rules of a node are used as the condition to determine when an action is allowed.

2. *Act*, is a non-empty set of action names. For each node and each input/output port pair, an action is created. When an action fires, it consumes data from its input data flows and generates data on its output data flows. Data from data stores can also be read or updated by the action when the node (the action is associated with) has read or write access to the data store.
3. *Init*, is the set of initial states. For each input port of a CDFD an initial state exists such that only data flows connected to the input port contain data, and each state is also restricted so that predicates specified in the *Init* process (part of a SOFL specification) are valid.
4. *Actions*, is a collection of action sets; each set of actions denotes the actions that are allowed to fire simultaneously for a state in the action system.
5. *Term*, a set of states in which the CDFD generates output. Whenever a state $\sigma_1 \in Term$ is reached in an execution path of an action system, it is not allowed for a state $\sigma_2 \in Term$ to exist in the path after σ_1 was seen. Also, each trace must contain a vertex contained in the set *Term*.
6. *N*, is the set of all vertices, where each vertex represents a state space.
7. *E*, is the set of all transitions between vertices labelled by elements in *Actions*, and each transition defines a predicate transform, i.e., transitions are labelled by a set of actions.

For an action system to be valid, there must be no read/write violation of the data stores, and the predicate transforms by element in *Actions* must create valid predicate transform, when all the predicate transforms in a trace are combined.

Algorithm 1 is extended to include the addition information of when data stores are being accessed, the predicate transforms defined, and when different type of nodes are allowed to fire to create an action system for a CDFD, see Algorithm 2. This action system is the “ideal” case of how data flow through a CDFD, i.e., “ideal” as in that the maximum amount of information is used to create the action system.

Data: A CDFD

Result: An action system $(Act, Var, Init, Actions, Term, N, E)$

Each input port of the CDFD defines an initial state space, these vertices are put into the set $Init$;

Restrict each state space in $Init$ such that the predicates defined by the $Init$ process and the invariants defined in the module are valid;

Put all the node in $Init$ on a stack S ;

Let the current node type be $process$;

for *pop the vertex v from the stack S* **do**

 Let I be the set of node that can fire in state space v ;

 Let C be a collection consisting of a set of nodes, where each set is a duplicate of I and a different combination of output port are marked to generate data;

if *current node type is process* **then**

 Remove all nodes from C that are not SOFL processes;

 Set the current node type so that the next iteration only *controlnodes* will fire;

else

 Remove all nodes from C that are not control nodes;

 Set the current node type so that the next iteration only *process* will fire;

end

for $c \in C$ **do**

 If data stores accessed by an action in c resulted in a read/write access violation, create an error, and mark the action system as invalid;

 Let e be the state when the set of nodes c transform the state space v , consume and generate the needed data;

if *an output of the CDFD can consume data from e* **then**

 Let ee be the state space where the data is consumed from e ;

 Add ee to $Term$;

 Add ee to to the stack;

 Add the transition $v \xrightarrow{c} ee$ to the set of edges (E) and tagged with the port consuming data;

else

 Add e to the set of vertices and the transition $v \xrightarrow{c} e$ to the set of edges (E);

 Add e to to the stack;

end

 Add c to $Actions$ and e to N and to S if e was not in N ;

end

end

Jump to Algorithm 3 and back;

Algorithm 2: Generation of an action system from a CDFD

for $A \in Actions$ **do**
 The predicate transform defined such that the state space

$$I \bigwedge_{a \in A} pre(A)$$
 is transformed into the state space

$$I' \bigwedge_{a \in A} post(A)$$
 where I and I' are the required invariants.;
end

Algorithm 3: Second part of Algorithm 2

3.3 Verification

All the conditions that need to be satisfied (as given in the previous section) must be “untangled” so that a separation can be made between static and dynamic semantics. The “untangling” is described here by defining an order in which the conditions must be verified. For example, Condition 9 needs to be verified before Condition 15, but Condition 9 does not provide any input for the computations to verify Condition 15. If Condition 9 is not satisfied, no meaning can be derived from the results obtained from verifying Condition 15. This allows the computation part of the verification to be mostly independent of each other, but not the interpretation of the results.

Verification of a specification is done by starting with the CDFD located on the endpoints of the hierarchical structure and by moving up the structure until the top most CDFD is verified. Each CDFD is verified independently of each other and then “stitched back together” by verifying refinement of each⁷ process refined by a CDFD, to recreate a similar “hierarchical structure”. When a CDFD is verified at the endpoints (of the hierarchical structure), it is assumed that all processes can be refined to an implementation.⁸ Figure 3.12 shows one of many allowed orderings for evaluating the CDFDs which are in a “hierarchical structure”.

Each CDFD in a specification is independently analysed and the steps that are needed to perform the verification are given by:

1	2	3	4	6	5	7
8	9	10	11	12	13	

The next step in the verification is that the hierarchical structure needs to be verified, and this is done by verifying the refinement of process by a CDFD. For a CDFD that refines a process:

14	15
----	----

⁷If such a refinement exists.

⁸This assumption is made since no more information is available in the specification that can be used to verify these processes.

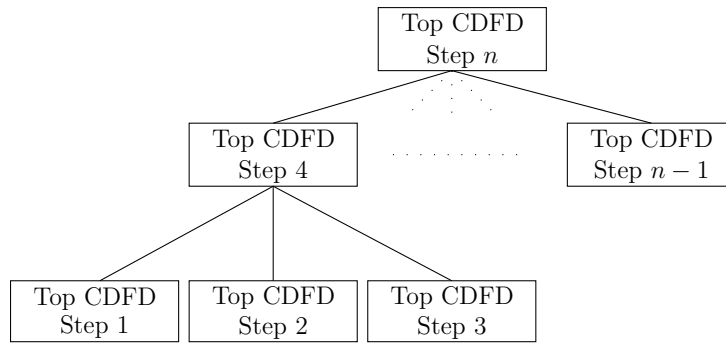


Figure 3.12: An example of verification order of CDFD in a “hierarchical structure” that consist of n CDFDs.

The above verification steps are automated by the developed tool, but the interpretation of each verification’s result is still a manual approach.

Chapter 4

Static Semantics

Verification of the static semantics uses information that is “fixed” in the specification. This chapter uses the semi-formal semantics of Section 3.2 and information that is available during the static semantics’ verification phase, to derive a set of tests (see Section 4.5) to determine if certain conditions defined in Section 3.2 are valid. These tests will be defined in terms of SMT-LIB so that an SMT-Solver (see Sections 4.3 and 4.4) can determine if the specifications satisfy the static semantics. Parsing of the dialect is not discussed in this chapter but in Chapter 6.

4.1 Type Checking

Type checking is used to determine if a module and CDFD are well-formed. Expressions and elements in a CDFD are given a type as well as the CDFD itself. When a module or CDFD is well-formed all the element and expression must have a well-defined type. An expression is defined using basic types (from [46]) where new types are introduced for element of a data flow diagram.

4.1.1 Basic Types

Basic types are defined in [46] and are reused almost without any modifications. A deviation is where a quantifier is used to “iterate” over a list of values and perform an evaluation of a sub-expression. This is in line with the idea that predicates also define types. The list consists of element of a fixed type, and a predicate is defined for each element. It was decided not to follow this route further in this dissertation.

The symbol \mathcal{T} will be used to denote a type and \mathcal{T}_b any basic type. When the type is boolean $\mathcal{T}_{\mathbb{B}}$ will be used.

4.1.2 Data Flow Types

Types are defined for element of a data flow diagram and are:

Data stores. Data stores have a basic type but when a process accesses a data store, read or write access is also associated with the data store. The type of read or write access is given by \mathcal{T}_{rw} .

Data Flows. Data flows are used to connect two ports to each other, but they do not depend on the connected ports to define their type. The type is given by

$$flow \langle \mathcal{T}_{\mathbb{B}}, \mathcal{T}_b, \mathcal{T}_{\mathbb{B}} \rangle : \mathcal{T}_F$$

where the first predicate type restricts the input data that the data flows accepts, the middle is a basic type that defines the type of the data value that is transported, and the last predicate defines a restriction of the type \mathcal{T}_b and define the allowed values from the data flows. The two predicates are taken to be the same as the pre and post condition of the connected ports. Inclusion of the predicate is more for convenience, since strictly they are redundant and can be derived from the CDFD and associated nodes.

The two predicates are used to add constraints on the values of type \mathcal{T}_b that are allowed, and both must be true. Let p_1 be the first predicate and p_2 the second predicate, then type is valid if

$$p_1 \longrightarrow_o p_2.$$

Ports. Ports are used as the input or outputs of data in nodes, and they are connected by data flows with each other. The type of a port is given by

$$port \langle \{\mathcal{T}_F\}, \mathcal{T}_{\mathbb{B}} \rangle$$

where the set of data flows types are the data flows connected to the port, and the predicate is the pre or post condition associated with the port.

A port is valid when the variables of the data flows connected to it define a space that is contained in the state space defined by the port's predicate.

Nodes. A node type is for computational elements in a CDFD as well as for a CDFD itself. A node consists of a list of input ports, list of output ports, and the data store that it accesses. A node's type is given by

$$node \langle [\mathcal{T}_F], \{\mathcal{T}_B \times \mathcal{T}_{rw}\}, [\mathcal{T}_F] \rangle$$

For each type of node there are additional constraints on the input and/or output ports.

Processes and CDFD: They must have at least one input and output port.

Condition structures: There is one input port and at least two output ports. Let p be the predicate of input port and q_i the predicate of the output ports, then it must be the case that

$$p \longrightarrow_o \bigvee_i q_i$$

and

$$q_i \wedge q_j = false$$

for each pair (i, j) of distinct values.

Merging structure: There is more than one input port and only one output port. Let I_i be the data types of the values received as input, ordered with the same ordering¹ as used in the syntax of the dialect, and O the type of the data generated by the output port. Then, it must be the case that

$$O = I_0 \times I_1 \times \cdots \times I_{n-1}$$

where n is the number of input ports.

Separating structure: There is more than one input port and more than one output port. The transformation applied to its input type is the inverse of the transformation applied by merging structures.

Broadcast structure: There is one input port and more than one output port. Also, the type of the input data is the same as the data type each of the output ports generate.

A data flow diagram defines how the node type of a CDFD is constructed by using node and data flow types. For a CDFD type to be valid:

- The first predicate of data flow type must evaluate to true, for all values it receives.
- A pre condition of any node must also evaluate to true, for all values it receives.

When determining if a CDFD type is valid, it is assumed that its pre conditions are true where used.

When a CDFD refines a process, the type of the CDFD must be compatible with the type of the process. The two types are compatible if the refinement conditions are satisfied.

Verification of types are done in two phases:

1. Verification of basic types, done by Xtext.
2. Verification by use of a SMT-Solver, done here.

4.2 Scope and Variable Definitions

Variables are defined in modules and input/output ports of nodes and function. Visibility of variables is also the same as in SOFL.

¹SOFL use the order in which the data flows appear in the visual representation of the CDFD, thus changing the layout of the diagram will change the meaning of the CDFD

4.3 Motivation for Use of SMT-Solver

Proving that static semantics is satisfied involves the use of correct types. In order to determine if types have been correctly used, state spaces created by predicates are required to be not empty, and where required, are included in another state space. Proving that the state spaces are valid according to the static semantics can be done using either interactive² theorem proving or an automated approach that only requires a question be asked and a result be obtained.

The automatic approach is the preferred solution in determining if logical formulæ are valid. A Satisfiability (SAT)-Solver is the tool that can be used, and it requires that all predicates be translated into a logic that contains only Boolean data types. This is inconvenient to use, but more importantly, a SAT problem is NP-complete³ which is not promising. A SAT problem is a predicate that contains variables for which an assignment of variables needs to be determined that will make the predicate true, i.e., the predicate is satisfiable.

An alternative is to use a SMT-Solver which allows the use of data types such as integers and the use of a fragment of its main⁴ logic. Use of a fragment of the main logic allows the use of algorithms that provide a specific advantage when solving problems in that logic. This does not guarantee that all problems are solved faster than using a SAT-Solver, but in some cases, faster solutions are found. The ability to use more data types (like integers) in predicate is not just a convenience, but also saves time when defining a problem to solve. It also decreases errors by not requiring data types to be translated each time a SAT problem is being defined.

In order to prove the properties of a design in the dialect, the conditions of Section 3.2 need to be written as a SMT problem. From the created SMT problem, it can be determined if a state space is empty and is contained in another state space. This directly addresses Conditions 5, 6 and 15, but addressing Conditions 8 and 10 is more involved and the approach taken is discussed in Sections 4.5 and 4.5.3. Most of the problems that need to be solved are of the form (also see Equation 3.1):

$$predicate(\varsigma) \longrightarrow_o predicate(\sigma)$$

with ς and σ state space where it needs to be proven that $\varsigma \neq \emptyset$ and $\varsigma \subset \sigma$. To prove that $\varsigma \neq \emptyset$ it is only needed that $predicate(\varsigma)$ is satisfiable. To prove that $\varsigma \subset \sigma$ let S be the set of all free variables in $predicate(\varsigma)$ and let

$$P = \forall_S (predicate(\varsigma) \longrightarrow predicate(\sigma))$$

where the only free variables of $predicate(\sigma)$ can be free variables in P and not free variables in $predicate(\varsigma)$. Thus the solution provided by the SMT-Solver will indicate

²For example of interactive theorem provers are Isabelle(<https://isabelle.in.tum.de/overview.html>) and Coq(<https://coq.inria.fr>)

³NP-complete is a complexity class that define problems which require an amount of resources bounded by an exponential function to solve the problem.

⁴From <http://smtlib.cs.uiowa.edu/logics.shtml>: “AUFLIA: Closed formulæ over the theory of linear integer arithmetic and arrays extended with free sort and function symbols but restricted to arrays with integer indices and values.”

that a valid state exist in σ whenever $predicate(\varsigma)$ is true, which verifies containment.

A number of SMT-Solver exist [2, 22, 61] which are all candidates for solving the SMT problems. Fortunately a common format, SMT-LIB, exists [3] which defines a standardised language in which problems that need to be solved can be formulated. Of the available options Z3 [61] was selected. Z3 has already been used to solve practical problems [9, 15] however there is no real reason not to use any of the other available SMT-Solver, given the scope of this dissertation.

4.4 SMT-LIB Usage

A logic is used for the SMT-Solver Z3 which allows quantifiers as well as a theory that allows integer and real data values. For ease of use, the solver will be asked to select the correct logic and theory that allows all the required elements, thus the input files for the solver will start with:

```
(set-option :print-success false)
(set-logic ALL)
```

The next part of the file is used to define all the variables that have global scope in the file, e.g.,

```
(declare-const var_int Int)
(declare-const var_real Real)
(declare-const var_bool Bool)
```

which define a variable `var_int` of integer type, a variable `var_real` of real type, and a variable `var_bool` of boolean type. Natural numbers are also needed, and are “created” by restricting the assignments that are allowed to the variables:

```
(declare-const var_nat Int)
(declare-const var_nat0 Int)

(assert (>= var_nat0 0))
(assert (> var_nat 0))
```

with `var_nat0` of type `nat0`, and `var_nat` of type `nat`. The `assert` statement is used to restrict assignments that are considered by the solver in order to determine if its input can be satisfied.

Binary operations are written in the form `(op (a b))` with `op` the operation to perform; the operation written infix notation is $a \text{ op } b$. An example of logical expressions that are used in the translation is given by:

Boolean operation: Example of expressions using boolean variables `var_bool0` and `var_bool1`

```
(assert (and
  var_bool0
  var_bool1))
(assert (or
```



```
var_bool0
var_bool1))
(assert (not var_bool0))
(assert (=>
var_bool0
var_bool1))
```

Arithmetic operation: Example using numerical types are:

```
(assert (>
(+ var0 var1)
var0))
(assert (>
(- var0 var1)
var0))
(assert (>
(* var0 var0)
var0))
(assert (not
(=
(div var0 var0)
0)))
```

Quantifier: Examples are:

```
(assert (forall (var_int Int)
(or
(>= int_var 0)
(< int_var 0))))

(assert (exists (var_int Int)
(>= int_var 0)))
```

where both quantifiers are defined over `int_var` and both formulæ are true. The variable over which the quantifier are defined is only defined for the scope of the quantifier, and can have the same name as a variable in the scope that contains the quantifier.

Only expressions that use numerical and boolean values are translated to SMT-LIB since there is a close relation between an expression on the dialect and SMT-LIB, a direct translation is used. The only exception is where `nat` or `nat0` is needed, then the SMT type `Int` will be used with a predicate that restricts allowed values so that only values of types `nat` or `nat0` are allowed.

4.5 Translations

The translation to SMT-LIB is used to verify some conditions of Section 3.3, and so part of the static semantics.

4.5.1 Refinement

This section defines tests that verify if Condition 15 is valid for process n , which is refined by a CDFD c . During refinement a CDFD is used as a refinement of process and pre and post condition of a CDFD, and the process is verified here for correct refinement. A CDFD can associate more than one port with a port in a process, a mapping \mathcal{G} is used to associated a port of a process with the associated port in the CDFD.

There is no syntax in the dialect to define this mapping explicitly, and the mapping \mathcal{G} is constructed by scanning port of the process and CDFD from left to right. A port in the CDFD that are connected to the same data flow as the ports in the process, are associated with each other. Note: only adjacent ports in a process can be associated with the same port in the CDFD and only one port in a process can be associated with one port in the CDFD.

The state spaces and predicates used during the verifications are defined as follows. For each input port i and output port o of the process:

1. The input and output state spaces of a process considered during refinement are:
 - (a) An input port with allowed state space σ , where the predicates that define the state are given by:
 - If the input port receive unmodified⁵ data from an input port of the CDFD, then the initialisation condition of the module is included in constructing the state space: $init_p$.
 - The invariant of the module in which the process is defined: inv_p .
 - The pre condition of the port is applicable: $pre(n, i)$.

Thus the predicate $predicate(\sigma)$ is defined by the conjunction of all the above predicates that are applicable.

- (b) An output port o with allowed state space σ' , where the predicates that define the state are given by:
 - The invariant of the module in which the process is defined: inv_p
 - The post condition of the port: $post(n, o)$.

Thus the predicate $predicate(\sigma')$ is defined by the conjunction of all the above predicates.

2. The input and output state spaces of a CDFD refining the process are:
 - (a) The input state space ζ associated with state space σ of the process, where the predicates that define the state are given by:

⁵The data can also be passed on via a control structure.

- The initialisation condition of the module in which the CDFD is defined: $init_c$.
- The invariant of the module in which the CDFD is defined: inv_c
- The pre conditions of the associated input ports of the CDFD:

$$\bigvee_{j \in \mathcal{G}(i)} pre(c, j).$$

Thus the predicate $predicate(\varsigma)$ is defined by the conjunction of all the above predicates.

- (b) The output state space ς' associated with the state space σ' of the process, where the predicates that define the state are given by:
- The invariant of the module in which the CDFD is defined: inv_c
 - The post conditions of the associated output ports of the CDFD:

$$\bigvee_{j \in \mathcal{G}(o)} post(c, j).$$

Thus the predicate $predicate(\sigma')$ is defined by the conjunction of all the above predicates.

3. It is not allowed for either of the state spaces σ , σ' , ς' or ς to be empty.

The predicate that needs to be verified for the “downward direction” is

$$predicate(\sigma) \longrightarrow_o predicate(\varsigma)$$

and written explicitly it is

$$(init_p \wedge inv_p \wedge pre(n, i)) \longrightarrow_o \left(inv_c \wedge init_c \wedge \left(\bigvee_{j \in \mathcal{G}(i)} pre(c, j) \right) \right) \quad (4.1)$$

when the initialisation is applicable the predicate is give by

$$(inv_p \wedge pre(n, i)) \longrightarrow_o \left(inv_c \wedge init_c \wedge \left(\bigvee_{j \in \mathcal{G}(i)} pre(c, j) \right) \right). \quad (4.2)$$

For the “upward directed”, the predicate used is

$$predicate(\varsigma') \longrightarrow_o predicate(\sigma')$$

and written explicitly it is

$$\left(inv_c \wedge \left(\bigvee_{j \in \mathcal{G}(o)} post(c, j) \right) \right) \longrightarrow_o (inv_p \wedge post(n, o)). \quad (4.3)$$

The Condition 15 is verified by formulæ 4.1, 4.2 and 4.3.

Using the example in Chapter 6, a script used to verify the “downward direction” output port 2 of the CDFD is given by:

```
(set-option :print-success false)
(set-logic ALL)
(declare-const parent_var Int)
(declare-const ss1 Int)
(declare-const ss2 Int)
(declare-const ss3 Int)(push)
(assert
  (and (and
    (> parent_var 10)(< parent_var 40))
    (or (and (> parent_var 10)(< parent_var 50)) true)))
(echo "Test if the state space being refined is not empty")
(check-sat)
(pop)
(push)
(assert(forall((parent_var Int))
  (=>
    (and(parent_var 10)(< parent_var 40))
    (and(and(and
      (> ss1 9)(< ss1 ss2))( < ss2 ss3))( < ss3 200))))))
(echo "Check that the lower invariant are satisfiable
  given that the upper invariants are true")
(check-sat)
(pop)
(push)
(assert (forall ((parent_var Int))
  (=>
    (and (and
      (> parent_var 10)
      (< parent_var 40))
      (or
        (and (> parent_var 10)(< parent_var 50)) true))
    (and (and (and (and
      (> ss1 9)(< ss1 ss2))
      (< ss2 ss3))( < ss3 200))
      (or true true))))))
(echo
  "Check if the lower state space is contained by the upper
  (port being refined)")
(check-sat)
```

(pop)

An additional test is possible using $pre(n, i, o)$ instead of $pre(n, i)$, and $post(n, i, o)$ instead of $post(n, i)$, in the above conditions. This will test if each execution path in the CDFD defined by pre and post condition pairs also satisfy the refinement obligation. These additional test are not required by the process being refined, since the process does not care if the CDFD uses an additional input and/or output port to satisfy the refinement constraints, or have unused ports. Despite not being strictly required by refinement, these additional test are useful to:

1. determine if a CDFD defined new ports that are not required by the process being refined, or
2. provide information of why the tests failed, if the first set of tests fail.

The predicate that needs to be verified for the “downward direction” is

$$(init_p \wedge inv_p \wedge pre(n, i, o)) \longrightarrow_o \left(inv_c \wedge init_c \wedge \left(\bigvee_{j \in \mathcal{G}(i)} pre(c, j, o) \right) \right)$$

when the initialisation is applicable the predicate is

$$(inv_p \wedge pre(n, i, o)) \longrightarrow_o \left(inv_c \wedge init_c \wedge \left(\bigvee_{j \in \mathcal{G}(i)} pre(c, j, o) \right) \right).$$

For the “upward directed”, the predicate used is

$$\left(inv_c \wedge \left(\bigvee_{j \in \mathcal{G}(o)} post(c, i, j) \right) \right) \longrightarrow_o (inv_p \wedge post(n, i, o)).$$

4.5.2 Invariants and Initialisation

Condition 5 is partially verified by the test defined here. The invariant inv is defined in a specification by the predicate following the keyword **inv** in the module, and the initialisation condition $init$ defined in the process **Init**.

The invariant inv must be true whenever a set of nodes do not fire, where the initialisation $init$ defines the largest state space in which the module can be initialised. Thus, for each initialisation the invariant must be satisfiable and is defined by

$$init \longrightarrow_o inv.$$

The translation for the predicates given by:

$$inv = (now_store < 10 \wedge store > 10)$$

$$init = (now_store < 0)$$

is given by:

```

(push)
(assert (< now_store 0))
(echo "Check if the environment is satisfiable")
(check-sat)
(pop)
(push)
(assert (forall ((now_store Int ))
  (=>
    (< now_store 0)
    (exists ((store Int ))
      (and (< now_store 10) (> store (- 10)))))))
(echo "Check if the environment is satisfiable")
(check-sat)
(pop)

```

4.5.3 Node State Space Transition

The tests defined here are used to verify Conditions 10 and 6, and partially verify Condition 5.

A node realises a transition between state spaces that is indicated by the variables before and after the node fires. For variable names before the node fires, the string $\mathbf{t_}$ is added to the name of the variable and refers to the state space before the node fires. By using predicate containing variables of the two state spaces, a relation between the state spaces is defined. For example, to define that a variable \mathbf{x} is increased by a transition, the post condition of the transition is defined by $\mathbf{t_x} < \mathbf{x}$.

For each process n with input port i and output port o , the predicate used in the verification is:

1. For all data stores that are accessible with read-only access an invariant I_{store} is created that ensures the data stores remain unchanged. For example, for a data store \mathbf{store} , the equation $(=\mathbf{t_store} \ \mathbf{store})$ will only allow state transitions where \mathbf{store} remains unchanged.
2. For starting invariant, I_{start} , if the input port is connected to an input port of the CDFD the initialisation constrain are also included, thus $I_{start} = inv \wedge init \wedge I_{store}$, otherwise $I_{start} = inv \wedge I_{store}$.
3. The final invariant, I_{end} , is the invariant of the module inv and predicates are added for the read only data stores accessible by this node.
4. The pre condition is the condition associated with the input port, $pre(n, i, o)$.
5. The post condition is the condition associated with the output port, $post(p, i, o)$.

For a node to be realisable the following predicate must be satisfiable for all input and output port pairs:

Condition 10 is verified by

$$I_{start} \wedge pre(n, i, o) \longrightarrow_o I_{end} \wedge port(n, i, o).$$

Condition 6 and 5 are partially verified by

$$pre(n, i, o) \longrightarrow_o I_{start}.$$

Condition 6 is partially verified by

$$post(n, i, o) \longrightarrow_o I_{end}.$$

For the process P_{loop} a translation to SMT-LIB for a port pair with predicates

$$\begin{aligned} init &= (ss3 < 20) \wedge (ss3 > 2) \\ inv &= (ss1 > 9) \wedge (ss1 < ss2) \\ &\quad \wedge (ss2 < ss3) \wedge (ss3 < 200) \end{aligned}$$

$$\begin{aligned} I_{store} &= t_ss2 = ss2 \\ I_{start} &= (ss1 > 9) \wedge (ss1 < ss2) \\ &\quad \wedge (ss2 < ss3) \wedge (ss3 < 200) \\ &\quad \wedge (ss3 < 20) \wedge (ss3 > 2) \\ &\quad \wedge (t_ss2 = ss2) \end{aligned}$$

$$\begin{aligned} I_{end} &= (ss1 > 9) \wedge (ss1 < ss2) \\ &\quad \wedge (ss2 < ss3) \wedge (ss3 < 200) \end{aligned}$$

$$\begin{aligned} pre(n, i) &= out_noloop < 30 \\ port(n, o) &= (d_data = true) \\ &\quad \wedge (d_data = (out_noloop < 40)) \end{aligned}$$

is given by:

```
(set-option :print-success false)
(set-logic ALL)
(declare-const d_data Bool)
(declare-const ss1 Int)
(declare-const ss2 Int)
(declare-const ss3 Int)
(declare-const t_out_noloop Int)
(declare-const t_ss1 Int)
(declare-const t_ss2 Int)
(declare-const t_ss3 Int)
```

```

(push)
(assert
  (and(and(and(and(and
    (> t_ss1 9)(< t_ss1 t_ss2))(< t_ss2 t_ss3))
    (< t_ss3 200))(= t_ss2 ss2))(< t_out_noloop 30)))
(echo "Check if pre condition and invariant is satisfiable")
(check-sat)
(pop)
(push)
(assert
  (and(and(and(and
    (> ss1 9)(< ss1 ss2))(< ss2 ss3))(< ss3 200))
    (and
      (= d_data true)
      (= d_data(< t_out_noloop 40))))))
(echo "Check if post condition and invariant is satisfiable")
(check-sat)
(pop)
(push)
(assert
  (forall((t_out_noloop Int))
    (=>
      (< t_out_noloop 30)
      (and(and(and(and
        (> t_ss1 9)(< t_ss1 t_ss2))(< t_ss2 t_ss3))
        (< t_ss3 200))(= t_ss2 ss2))))))
(echo "Check that the invariant is satisfiable
  given the pre condition is true")
(check-sat)
(pop)
(push)
(assert
  (forall((d_data Bool)(t_out_noloop Int))
    (=>
      (and
        (= d_data true)
        (= d_data(< t_out_noloop 40)))
      (and(and(and
        (> ss1 9)(< ss1 ss2))
        (< ss2 ss3))(< ss3 200))))))
(echo "Check that the invariant is satisfiable
  given the post condition is true")

```



```

(check-sat)
(pop)
(push)
(assert
  (forall((t_ss3 Int)(ss2 Int)(t_out_noloop Int)(t_ss2 Int)(t_ss1 Int))
    (=>
      (and(and(and(and(and
        (> t_ss1 9)(< t_ss1 t_ss2))(< t_ss2 t_ss3))(< t_ss3 200))
        (= t_ss2 ss2))(< t_out_noloop 30))
      (and(and(and(and
        (> ss1 9)(< ss1 ss2))(< ss2 ss3))(< ss3 200))
        (and(= d_data true)(= d_data(< t_out_noloop 40))))))))))
(echo "Check if the P -> Q is satisfiable")
(check-sat)
(pop)

```

4.5.4 Execution Path Approximation

An execution path of the action system defines a sequence of predicate transforms that are used to transform input of a CDFD to its output. Here an approximation is created for the sequence of predicate transforms since only information available during static analysis can be used.

For such a sequence of predicate transforms to be valid, each predicate transform must only be performed in a valid state which requires its pre condition to be true. For an execution path

$$\dots \xrightarrow{A_{i-1}} \sigma_i \xrightarrow{A_i} \sigma_{i+1} \xrightarrow{A_{i+1}} \dots$$

in the action system with A_i the set of action resulting in the transition and for state spaces σ_i , then for the execution path to be valid it is needed that $\sigma_i \subseteq \mathbf{wp}(A_i, \sigma_{i+1})$ which would require exact knowledge of A_i to verify. Exact knowledge of A_i is not available during static verification and approximation will be used by considering the actions connected to the input of each action in A_i . This approximation will also determine the limitations of the static verification.

For each action $a_j \in A_i$ the actions connected to its input are given by $B_i = \{b_k\}_k$, the set B_i is used to approximate the state σ_i and is called ς_i (see Figure 4.1) and B_i is called the pre-image of a_i . Since B_i does not necessarily include all the actions that fire immediately before the set A_i , it leaves the possibility open that an action not in B_i influences a data store accessed by a_j thus making the approximation not valid.⁶ For each action $b_k \in B_i$, its contribution to ς_i is approximated by $pre(b_k)$. By only using the predicate $pre(b_k)$, not all the static information is included in the specification when

⁶Good invariants for the module might prevent this type of invalidation.

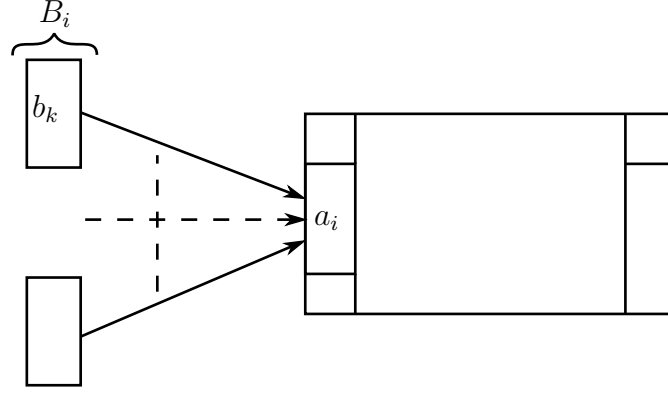


Figure 4.1: The pre image of an action a_i .

defining the end state of b_k . It will always be the case that the pre condition P_{b_k} of the node that contain b_k must have been true before b_k fires, and therefore will also be included.⁷ The contribution of $b_k \in B$ to the state space is now approximated by

$$pre'(P_{b_k}) \wedge post(b_k)$$

where the prime ' rename all the variables in the predicate to its previous state names, and

$$\varsigma_i = \bigwedge_k [pre'(P_{b_k}) \wedge post(b_k) \wedge I_{b_k}]$$

where I_{b_k} is the invariant applicable to the output of b_k . The approximated input state space of A_i is now given by

$$\omega_j = \bigcap_{a_k \in A_i} \varsigma_k$$

and the execution path is approximated by

$$\dots \omega_i \xrightarrow{A_i} \omega_{i+1} \xrightarrow{A_{i+1}} \dots$$

where it is required that

$$\omega_i \subseteq \mathbf{wp}(A_i, \omega_{i+1})$$

and is verified by validating for each a_i that

$$predicate(\omega_i) \longrightarrow_o pre(a_j) \wedge I$$

and I is the invariant applicable to the input of a_i ; a_i is an action associated with the

⁷This including of this “history information” does use dynamic semantics but is available during static analysis.

node n with input port i and output port o , and translates to

$$\bigwedge_k [pre'(P_{b_k}) \wedge post(b_k) \wedge I_{b_k}] \longrightarrow_o pre(n, i, o) \wedge I \quad (4.4)$$

and since it is not known during static verification which output port will generate data

$$\bigwedge_k [pre'(P_{b_k}) \wedge post(b_k) \wedge I_{b_k}] \longrightarrow_o pre(n, i) \wedge I. \quad (4.5)$$

The invariants I and I_{b_k} are either $int \wedge init$ or inv , depending on whether the initialisation condition is applicable to the port or not. Condition 8 is validated by using equations (4.4) and (4.5).

A translation is given below for process P_{second} in the small example, with the predicates used in the verification given by

$$\begin{aligned} I &= I_{first} = (ss3 < 20) \wedge (ss3 > 11) \\ pre(P_{first}) &= ss3 > 11 \\ post(P_{first}) &= (other_bottom > 10) \\ &\quad \wedge (to_second < 20) \\ &\quad \wedge (to_second > 10) \\ pre(P_{second}) &= (to_second < 20) \\ &\quad \wedge (to_second > 9) \end{aligned}$$

where $other_bottom$ is of type `nat0` and the translation is:

```
(set-option :print-success false)
(set-logic ALL)
(declare-const other_bottom Int)
(declare-const ss1 Int)
(declare-const ss2 Int)
(declare-const ss3 Int)
(declare-const t_ss3 Int)
(declare-const to_second Int)
(push)
(assert(>= other_bottom 0))
(assert
  (and(and(and(and
    (> other_bottom 10)(< to_second 20))
    (> to_second 10))
    (and(and(and
      (> ss1 9)(< ss1 ss2))(< ss2 ss3))
      (< ss3 200))))(> t_ss3 11)))
(echo "Check if environment is satisfiable")
(check-sat)
```

```
(pop)
(push)
(assert(>= other_bottom 0))
(assert
  (forall(
    (t_ss3 Int)(other_bottom Int)(ss3 Int)
    (to_second Int)(ss1 Int)(ss2 Int))
    (=>
      (and(and(and(and
        (> other_bottom 10)(< to_second 20))
        (> to_second 10))
        (and(and(and
          (> ss1 9)(< ss1 ss2))(< ss2 ss3))
          (< ss3 200)))(> t_ss3 11))
        (and(< to_second 20)(> to_second 9))))))
(echo "Check if the environment allows the port state")
(check-sat)
(pop)
```

Chapter 5

Dynamic Semantics

The action system given in Section 3.2 is used to create a dynamic semantics, by creating a translation using the language of mCRL2 to give a process algebra description. This translation only considers some information in the action system and creates an approximation of the semantics. Each CDFD is translated separately, and each translated CDFD is analysed separately.

The translation of an action system (see Section 3.2.4) is used to create a process algebra description, and the Label Transition System (LTS) (of the process algebra) is used to create a correspondence with the action system. Section 5.2 motivates why a process algebra is used (specifically mCRL2) to describe the behaviour of the CDFD. The action system is modified so that the correspondence with the behaviour of a CDFD, described by a process algebra description, is clear, see Section 5.5.

Only an approximation is created from the action system as the presence of data values in data flows is indicated by the presence of tokens, and no data values of a specification are used in the translation. Before the dynamic semantics can be verified, the following steps are needed:

1. Create a mCRL2 description so that a LTS that is an approximation of the action system defined in Section 3.2.4 is created (see Section 5.5).
2. Create modal formulæ [7] that use the created LTS as Kripke to prove properties that exist in the LTS (see Section 5.6).

When a LTS cannot be created it indicates that data stores accessed are invalid. If the LTS could be created, further validation is performed by the modal formulæ.

5.1 Firing Rules

An “execution” path is created by a set of nodes that fire in an organised manner, and is shown in Figure 5.1. For the nodes to fire in an organised manner, an element is needed to determine when the nodes are allowed to fire and to keep track of access to the data stores.

The sequence of events needs is:

1. The CDFD puts data on all input data flows connected to one of its input ports.

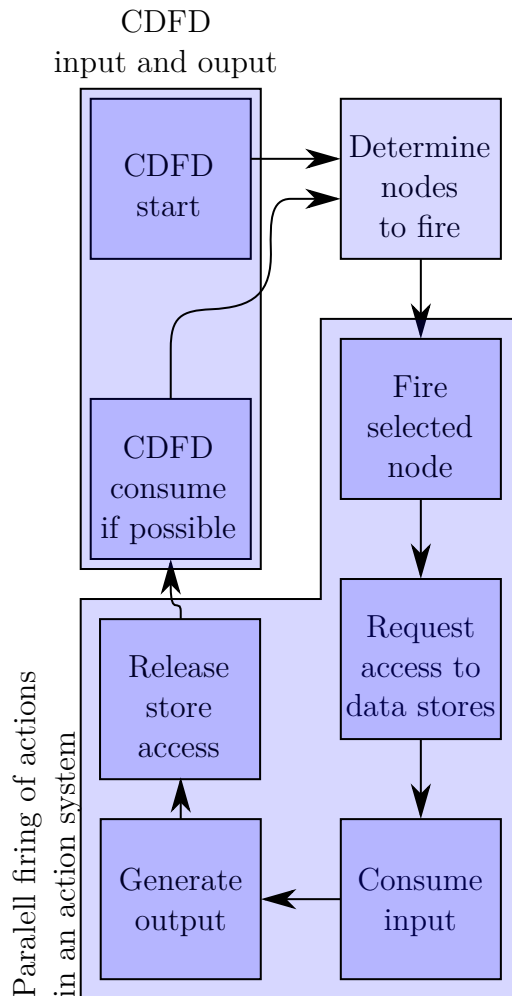


Figure 5.1: The sequence of events that must be allowed after communication of actions.

2. Start in a state where control nodes are allowed to fire.
3. Continue until there is nothing more to do:
 - (a) Determine the set of nodes that are allowed to fire.
 - (b) Allow the nodes to fire and keep track of data store access.
 - (c) Verify data store access.
 - (d) Change state to next type of nodes that are allowed to fire.
 - (e) The CDFD generates output on one of its output ports.

This allows the possibility for a CDFD to generate output multiple times on any output port. This allows the approximation of the action system to detect invalid CDFD that generate data on multiple output ports, by using a modal formula.

5.2 Motivation for Process Algebra

Process algebra is a standard method used to model the behaviour of concurrent systems. A number of different process algebras exists where the algebraic rules are used to manipulate a process algebra description are different. From all the available process algebras, mCRL2 [28] was used in this dissertation based on a recommendation.¹ The formal manner in which the semantics of the process algebra is defined gives considerable confidence in the result obtained by using the process algebra.

A process algebra is defined by actions and operations. Actions are events that are of interest and now only two operations: choice $+$ and sequential \cdot , are of interest as well as some axioms.² These are given below:

$$\begin{aligned}
 x + y &= y + x \\
 x + (y + z) &= (x + y) + z \\
 x + x &= x \\
 (x + y) \cdot z &= x \cdot z + y \cdot z \\
 \tau \cdot x &= x
 \end{aligned}$$

where x , y , and z are action and τ is a special action called an internal action. Later it will become clear what the additional meaning of an internal action is. These axioms are used as algebraic rules to simplify (or factorise) a process algebra description. The behaviour of process algebra is defined by the “order” of action that are allowed and can be described using trees, see Figure 5.2. The two trees are defined by the two processes

$$\begin{aligned}
 x \cdot (y + z) \\
 x \cdot y + x \cdot z
 \end{aligned}$$

which are not equivalent given the above axioms. Such trees were also considered in [60] where the focus was on when trees are equivalent and so defining a notation of “computation”, where the equivalence relation use is bi-simulation.³ To determine if two trees are equivalent, an equivalence relation is selected and the trees are equivalent only if the equivalence relation cannot distinguish between the two.

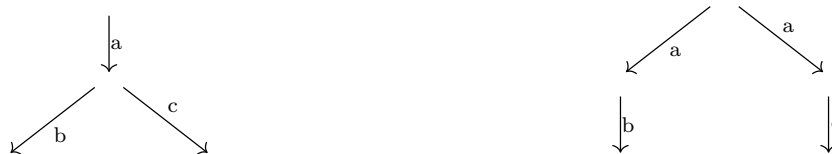


Figure 5.2: Tree of behaviours, used to illustrate when behaviours are equivalent.

¹Markus Roggenbach.

²This is a sub set of the axioms of mCRL2.

³For a related equivalence relation used in this dissertation, see Definition 3.

Two trees can be seen as equivalent by using a different equivalence relation (see Definition 4) which considers all paths from a trees root to its leaves. For these two trees the paths are given by

$$\{ab, ac\}$$

and on axiom level this would introduce the rule

$$x \cdot (y + z) = x \cdot y + x \cdot z.$$

This axiom is not part of mCRL2 when equivalent processes are defined, but will be used to change the meaning of the mCRL2 description in order to obtain the desired behaviour to describe a CDFD. The change in meaning does not “break” the process algebra description, it rather forces a translation that creates a new process algebra description that is no more equivalent (using weak bi-simulation) to the original process algebra description.

One of the main reasons for using a process algebra is that action are allowed to “run” in parallel. The axiom of a parallel operation is given by

$$x \parallel y = x \parallel y + y \parallel x + x \mid y$$

in mCRL2, where \parallel , and \mid are further auxiliary operations used to define parallel operations. For the purpose of this dissertation it is not necessary to consider further how a parallel operation is defined. Whenever processes “run” in parallel, the trees representing the process are put next to each other to create a directed graph. The actions of the “trees” occur independently of each other. Later, a more formal approach is defined (see Definition 2) where the trees are properly merged when the directed graph is created.

mCRL2 also define data types and allow actions to take parameters that are values of these data types. This requires the additional axioms:

$$\begin{aligned} \sum_{d:D} x &= x \\ \sum_{d:D} X(d) &= X(e) + \sum_{d:D} X(d) \\ \sum_{d:D} (X(d) + Y(d)) &= \sum_{d:D} X(d) + \sum_{d:D} Y(d) \\ \left(\sum_{d:D} X(d) \right) \cdot y &= \sum_{d:D} X(d) \cdot y \end{aligned}$$

where x and y are actions; X and Y are actions that take a data value as parameter; D is a data type, and the sum is over all the values of the data type, d . Conditional statements that substitute one of two process algebras descriptions in its place, depending on the value of the predicate, are also allowed. A product type can also be created from data types. All parameters are product types, therefore it is only needed to define actions that take no or one parameter.

Operations on process are also defined in mCRL2 and these are described using the

idea of representing a process by a tree.

Allow/block action by allowing some actions all other actions are blocked and vice versa. By blocking an action, it is equivalent to removing all sub-trees that follow the blocked action and the blocked action. When multiple actions are blocked the order in which the actions are blocked does not matter.

Hide by hiding an action, the action is redefined as an internal action. Internal actions are used by some equivalence relations and treat internal action as being of no interest, and can be seen as “filtering” internal actions. For example, by defining a as an internal action in $a \cdot (b + c)$ and “filter” the internal action out, the resulting process is $b + c$.

Communication when processes are combined using the parallel operation their event can occur independently of each other. By communication, actions in different trees are synchronised. Communication defines a new action, and whenever the actions that are synchronised occur at the same time, the new action is used. When actions that take a parameter are synchronised, the value of the parameter must also be equal and the same parameter value is used for the newly created action.

The language mCRL2 also includes a tool-set [16] that allows analysis of the process algebra description. Two important analyses that are used in this dissertation are: determination if a specific⁴ action occurred, and evaluation of modal logic formula (see Section 5.6). Use of modal formulæ allows properties to be described that are needed to relate different “state spaces” of a CDFD. Practical problems were solved using mCRL2 in [68, 71].

5.3 mCRL2 usage

The behaviour of an action is described by algebraic process theory which uses the tool-set mCRL2⁵ that extends a process algebra based on Algebra of Communicating Processes (ACP). This extension allows the use of data values when a description is created. Snippets from the created translation will be used to explain how the translation is created, and the syntax used by mCRL2 to describe a process.

Data types, namely sorts and basic types, are already defined in mCRL2. For a translation, custom types are created using enumeration types:

1. Port identification: `sort FlowId=struct flow1|flow2|...|flown;`
2. Data flow identification: `sort FlowId=struct flow1|flow2|...|flown;`
3. Node identification: `sort NodeId=struct node1|node2|...|noden;`
4. Node type identification: `sort NodeType=struct type_control|type_process;`

⁴For example an action that indicate some error state.

⁵www.mcr12.org

5. Keep track of the next type of nodes to fire:

```
sort Phase=struct exec_control|exec_process;
```

6. Data flow identification: `sort FlowId=struct flow1|flow2|...|flown;`

7. Data store identification and requesting read or write rights? for the data store identification:

```
sort StoreId=struct s0|s1|...|sn;
```

8. Data store access rights and the phase access:

```
sort Rights=struct read_access_begin|read_access_done|
write_access_begin|write_access_done|rights_done;
```

9. A mapping type

```
sort StoreMap = StoreId -> Int;
```

to keep track of how many SOFL processes access a data store. A separated instance of the mapping is used for read and write access.

Events are called actions in mCRL2 and are the observable events of a process. A tree can be used to represent a process with the edges labelled by the action names. An action is defined by

```
act
node_flow_consume : FlowId;
flow_consume : FlowId;
fire_ports : NodeId#PortId#PortId;
```

where the action `node_flow_consume` accepts a value of data type `FlowId` and multiple parameters by creating a product type of the parameters `NodeId#PortId#PortId`.

A process can consist of a single action but is usually created by combining the action with an operation.

Sequential composition is where the first action occurs before the following process.

```
act1.act2.(acp process);
```

Choice creates a branch in the tree (see Figure 5.3) describing the process for each action involved in the choice composition. A choice between two action is defined by `act1 + act2` and multiple actions by using

```
sum i:FlowId.node_flow_consume(i);
```

where the choice is between the actions `node_flow_consume` taking a different value as parameter.

Parallel composition of processes are defined by `process1||process2` to show that the processes run in parallel.

Branch condition allow the creation of a branch in the branch that is data dependant, for `f_set` a set type and use “is the set empty’ and the branch condition is given by:

```

(f_set == {}) ->
  true branch goto acp process
<>
  false branch goto acp process;

```

Definition of a process is given by

```

NodeFlowIdConsume(ndid:NodeId, f_set : FSet(FlowId), pid:PortId) =
  (f_set == {}) ->
    skip.NodeGenerate(ndid, pid)
  <>
    sum f: FlowId . (f in f_set) ->
      node_flow_consume(f).NodeFlowIdConsume(
        ndid, f_set - {f}, pid);

```

The behaviour of a process is illustrated in Figure 5.3 using a tree structure. The behaviour can also be described using a LTS (see Definition 2), where the states are used for the tree nodes and the transitions to indicate when an action occurred. The initial state is the root of the tree and the terminating states are those where no action is allowed to create a transition. The advantage of using a LTS is that a loop can be represented by a LTS with a finite number of states, whereas a tree would be infinite. The idea of a tree is useful for discussion purposes, but an LTS can always be created.

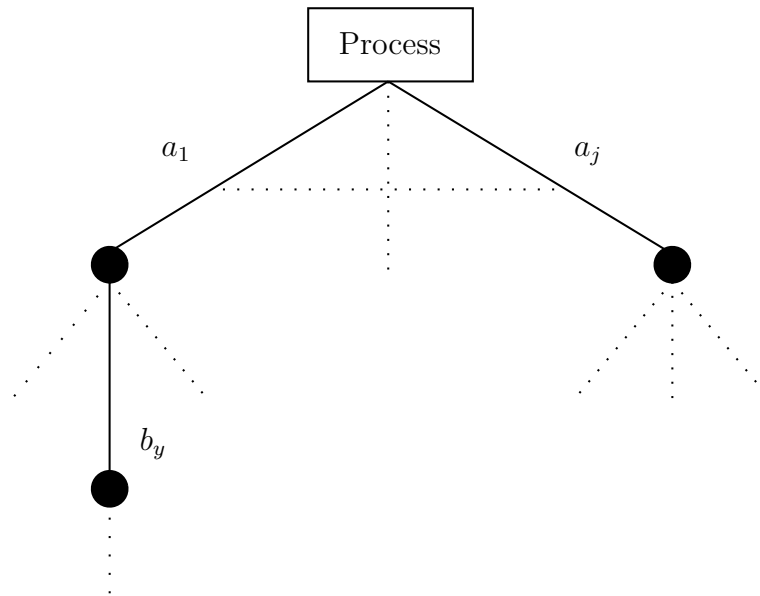


Figure 5.3: A tree structure of a process algebra description with actions a_i 's and b_j 's.

Definition 2 (Labelled Transition System). A Label Transition System (LTS) is a five tuple $A = (S, Act, \longrightarrow, sT)$ where

1. S is a set of state.
2. Act is a set of actions.

3. $\longrightarrow \subseteq S \times Act \times S$ is a transition relation.
4. $s \in S$ is the initial state.
5. $T \subseteq S$ is the set of terminating states.

It is common to write $t \xrightarrow{a} t'$ for $(t, a, t') \in \longrightarrow$.

There are also operations that allow manipulation of a process's actions to create a new process:

Allow actions accepts a process description and only allow the list of supplied actions
`allow({act1, act2}, acp process)`

Communication between action are defined by

```
comm({
  ...
  act1 | ... | actn -> synchronised_action_name,
  ...
}, acp process)
```

where the actions on the left-hand side of `->` defines a list of actions that occur in parallel; a new action name is created (on the right-hand side of `->`). If the actions take parameters, the parameters of all actions must be equal, and a match of the data values is also needed to communicate.

Rename of an action

```
rename({act1 -> new_name1, act2 -> new_name2, ...}, acp process)
```

Hiding of an action makes it unobservable in the resulting process

```
hide({act1, ..., actn}, acp process)
```

An action that is hidden is called an internal action τ , and it cannot take part in a communication between actions.

An example of how to use these operations is:

```
allow(
  {sact},
  ,comm({
    act1 | act2 -> sact,
  },
  sum d: Int. act1(d) || act2(d))
```

and the resulting process is `sum d: Int. sact(d)`. This pattern is used in the translation to communicate data values between processes and only allows the actions that communicate i.e., not allowing `act1` or `act2`.

5.4 Process Algebra Description

An action system is translated into mCRL2 processes; the processes and the communication between the processes are illustrated in Figure 5.4.

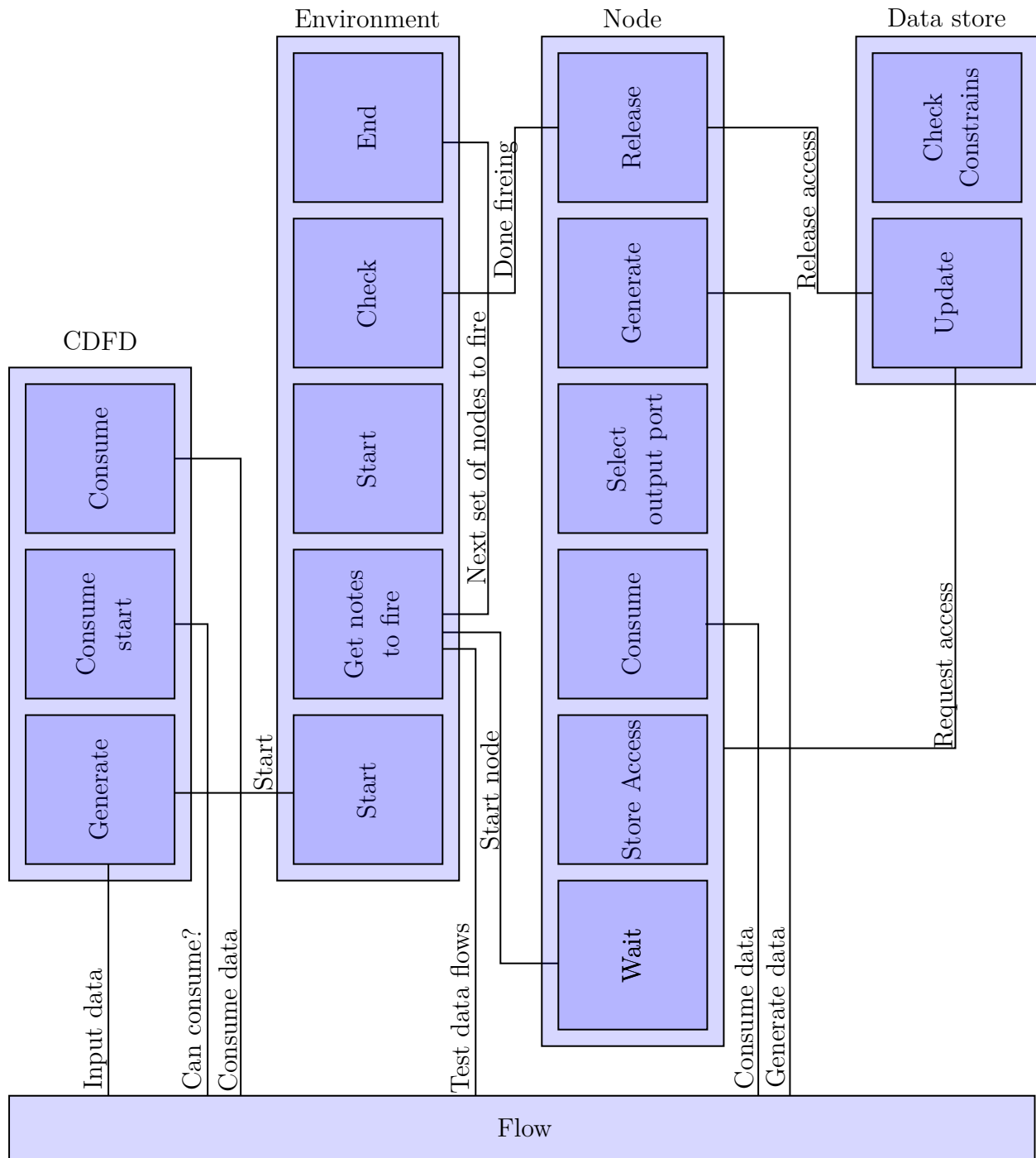


Figure 5.4: ACP processes used to specify a CDFD

Each mCRL2 process runs in parallel, and each process emulates a specific behaviour of elements in a CDFD. These processes do not run independently and communicate with each other in order to:

1. Make sure the behaviour of the mCRL2 processes is consistent with the behaviour of the CDFD that is translated.
2. Exchange data between processes. For example, to indicate which data stores are accessed and whether the data store is accessed with read or write access.

The actions created by the communication are defined as internal actions, and the original actions are blocked. The purpose of the actions that communicate is either to exchange data or act as synchronisation points. For example, a synchronisation point allows all nodes that fire to finish the fire phase before the next set of nodes that can fire are determined.

The mCRL2 processes and their sub-phases are:

CDFD This process is responsible for the input and output of the CDFD and consists of the following phases:

CdfdGenerate puts a token on the data flows that are connected to the selected port; the next phase is CdfdConsumeStart.

CdfdConsumeStart determines the output port of the CDFD that can consume tokens. If an output port of an CDFD can consume tokens, the next phase is CdfdConsume.

CdfdConsume consumes a token from the data flows connected to the output port of the CDFD that can consume data and was selected in the current process; the next phase is CdfdConsumeStart.

Repeated consumption allowed in order to detect designs that are invalid.

Environment This process is responsible for starting the firing of the nodes.

EnvironmentFirst wait for an input port of the CDFD to put token on data flows before making transition to the next phase, the phase Environment.

Environment determines the set of nodes that fire next based on the data flows that contain tokens. The nodes that are allowed to fire alternate between the processes and the set of structure nodes types, next phase EnvironmentStart.

EnvironmentStart fire each node determined in the previous step to be allowed to fire, next phase EnvironmentStartCheck.

EnvironmentStartCheck wait for each node to complete its “firing steps” before transition to the next phase EnvironmentEnd.

EnvironmentEnd set the state to indicate the next type of node that is allowed to fire, after all the nodes complete their “firing steps”, next phase Environment.

Environment Store This process keeps track of when nodes access data stores and the rights requested by the nodes. Whenever there is a read/write violation, an error action is created indicating the store where the violation occurred.

EnvironmentStore update the data structure used to keep track of the data stores accesses with read/write access, next phase EnvironmentStoreConstrain.

EnvironmentStoreConstrain determine if there is read/write access violation and create an error action if there is an access violation, next phase EnvironmentStore.

Environment Flow This process keeps track of which data flows contain tokens.

Node Each instance of this process represent a SOFL node. This process is responsible for:

Node wait for the node to be allowed to fire, and when allowed move to the next phase: NodeStoreAccess.

NodeStoreAccess request access to all data stores that this node accesses and select the input port from which tokens will be consumed. The next phase is NodeFlowIdConsume.

NodeFlowIdConsume consume token from all data flows connected to the selected input port, next phase NodeGenerate.

NodeGenerate select the output that will generate a token for each data flow connected to it, next phase NodeFlowIdGenerate.

NodeFlowIdGenerate generate token on all data flows connected to the selected output port, next phase NodeStoreRelease.

NodeStoreRelease release all the data store to which access was requested, next phase Node.

For each node in a CDFD, an instance of the “Node” process is created to describe the behaviour of the node (in the CDFD), and only one instance is created of the other mCRL2 processes. These processes are “run” in parallel, and the communication used to create a single process is described below.

Communication between the above processes is defined as follows:

1. Execution of the CDFD starts:

```
env_cdfd_start | cdfd_cdfd_start -> cdfd_start
```

processes that are synchronised “CDFD” and “Environment”.

2. A node starts its “firing step”:

```
env_node_start | node_execute -> start_fire
```

Processes that are synchronised “Environment” and “Node” where the “Environment” creates a list of nodes that are allowed to fire next, and only one such synchronisation occurs for each node in the list.

3. Request/Release access to data stores:

```
env_store_update | node_store_update -> store_update
```

Processes that are synchronised “Environment Store” and “Node” where each “Node” has a list of data stores to which it requires either read or write access. This synchronisation allows the process “Environment Store” to determine if there are any read/write access violation.

4. A node’s “firing step” is completed:

`env_node_end | node_end -> done_fire`

Processes that are synchronised “Environment” and “Node” where the process “Environment” waits for all “Node” processes to complete their firing phase and this synchronisation creates the waiting point.

5. Use to indicate the last step of a node’s “firing step” is complete:

`env_store_check | store_store_check | node_store_check
->store_check`

Processes that are synchronised “Environment”, “Environment Store” and “Node” where the process “Environment Store” updates its counter for each node that accesses a data store so that a new set of nodes are allowed to access data store independently of the current set.

The mCRL2 description needs to be transformed so that only the transitions, as shown in Figure 5.1, are allowed. This is done by blocking all the actions that are synchronised, but not the new action, e.g., for

`act1 | act2 | ->new_act`

actions `act1` and `act2` will be blocked but not the action `new_act`. Next the new action (`new_act` in the example) will be defined as an internal action. The last step will be to use weak bi-simulation (see Definition 3) as an equivalence relation when the new process is created. This will “filter” the internal actions, thus reduce the size of the LTS and “lock in” the behaviour as in Figure 5.1.

Definition 3 (Weak bi-simulation). Consider two LTS $A_1 = (S_1, Act, \longrightarrow_1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow_2, s_2, T_2)$. A relation $R \subseteq S_1 \times S_2$ is called a weak bi-simulation if for all $s \in S_1$ and $t \in S_2$ such that sRt , the following holds:

1. If $s \xrightarrow{a}_1 s'$, when
 - (a) either $a = \tau$ and $s'Rt$, or
 - (b) there is a sequence $t \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 \xrightarrow{a}_2 \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t'$ such that $s'Rt'$.
2. Symmetrically if $t \xrightarrow{a}_2 t'$, when
 - (a) either $a = \tau$ and sRt' , or
 - (b) there is a sequence $s \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 \xrightarrow{a}_1 \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ such that $s'Rt'$.
3. If $s \in T_1$, then there is a sequence $t \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t'$ such that $t' \in T_2$.
4. Again symmetrically, if $t \in T_2$, then there is a sequence $s \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ such that $s' \in T_1$.

The process algebra description is now ready to address the parallel behaviour of nodes that fire simultaneously on a CDFD.

5.5 LTS of Action System

The description in mCRL2 is translated to a LTS that describes the behaviour of the action system (see Section 3.2.4). This section describes how an action system is modified to a LTS as well completing the translation from the mCRL2 description.

Sequential Execution Model

mCRL2 allows parallel processes which are used to “run” the processes in Figure 5.1 in parallel, and this prevents the use of the process algebra’s parallel operation to allow SOFL processes of the dialect to fire in parallel. Thus, a sequential execution model [1] will be used to described execution paths.

In the action system, a set of actions fire simultaneously, but for a sequential model, only one action can fire at a time. The desired behaviour is obtained by marking the start and end of a set that fires and connecting the two markings with all possible ordering for the set of actions. An execution path only considers the orderings and will ensure only one action fire at a time. The same state for the end marking is ensured by:

1. The read/write mutual exclusion principle of data stored.
2. Each action in the set of actions is described by the name of the node, the input port that consume data, and the output port that generates data. Whenever data is consumed from a port or generated on a port, it does not influence any other actions (of nodes) in the set that currently fires.

This gives a sequential semantics of the nodes that fire in parallel, but requires that for each set of size n , all possible ordering of the set are considered. There are $n!$ orderings of the actions in the set. This grows faster than an exponential function which will unfortunately limits the analysis to only “small” CDFDs.

The start, end marking and action in the action system cannot be described by actions in a process algebra. How additional control information and other information is included is described below.

Sequential Traces and Action System

For the sequential execution model to be realisable by a process algebra, all the different orderings of actions (of nodes) that fire need to be considered equivalent. This equivalence of actions that fire does not require any special attention in creating the LTS. For the behaviour to be related to an action system of a CDFD, the LTS must use the execution path of the actions as its equivalence relation. Thus, both trees in Figure 5.2 must be equivalent. This is done using a weak trace equivalence relation, see Definition 4.

Definition 4 (Weak trace equivalence). Let $A = (A, Act, \longrightarrow, s, T)$ be a labelled transition system. The set of weak traces $WTrace(t)$ for a state $t \in S$ is the minimal set satisfying:

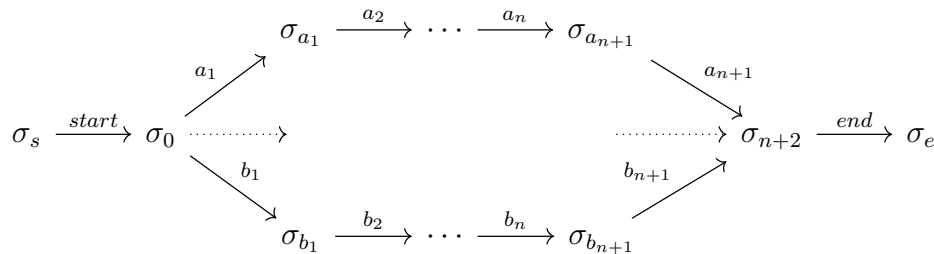
1. $\epsilon \in WTrace(t)$, it contains the empty trace
2. $X \in WTrace(t)$ if and only if $s \in T$, X indicates termination

3. if there is a state $t' \in S$ such that $t \xrightarrow{a} t'$ ($a \neq \tau$) and $\sigma \in WTrace(t')$ then $a\sigma \in WTrace(t')$.
4. if there is a state $t' \in S$ such that $t \xrightarrow{\tau} t'$ and $\sigma \in WTrace(t')$ then $\sigma \in WTrace(t')$.

Two states, $t, u \in S$ are called weak trace equivalent if and only if $WTraces(t) = WTraces(u)$. Two transition systems are called weak trace equivalent when their initial states are weak trace equivalent.

The action system also needs to be extended so that it “looks” more like the LTS of the process algebra description.

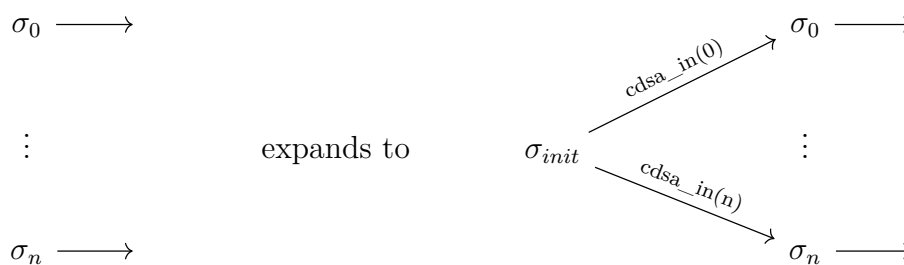
Sequence of actions that fire: In an action system, a set of action is allowed to fire simultaneously, and in the sequential execution model only one action is allowed to fire at a given time. An illustration is given below of how an action system is extended to include all possible combinations of actions (of nodes) that fire simultaneously:



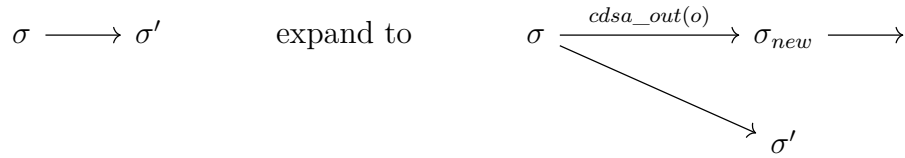
where states are given by σ_i and the labels a_i and b_i indicate different ordering of the actions in each trace. The two actions “start” and “end” indicate the start and end of the set of parallel actions.

The sequences of node actions that fire will exist in the LTS where the difference was addressed below.

Initial transitions: A LTS has a single initial state where an action system has a starting point for each input port in the CDFD, thus for the action system, an additional initial state is introduced and actions $\{cdsa_in(i)\}$ (with i an input port index) indicate transition from the starting state ‘to’ one of the starting states of the “original” action system, i.e.,



CDFD output transitions: For each state in the action system where output for the CDFD is generated, a new state and action $cdsa_out(o)$ (with o an output port index of the CDFD) is created. The new action indicates that the CDFD generates output data. Thus:



where σ is the original state and σ_{new} is the state after the output was consumed from the state σ .

Note: this expansion is not allowed for states between a start and end marker. For a set of action (of nodes) that fire, each node must first complete its firing before the CDFD can generate output.

Read/write violations: When there is a read/write access violation, the action system is extended to have a transition labelled with an error message, and the rest of the action system continues as usual after this transition. The LTS will have the same behaviour.

The tool `ltsconvert` of mCRL2 reduces the size of a LTS by using an equivalence relation. Additional information are also added to LTSs mainly for debugging purposes, and the list of actions is given in Table 5.1. Not all information in an LTS is always used. For example, when it is determined if a CDFD can generate output on a given port given that a specific input port initiates the execution, all the nodes that fire are not of any interest. After the LTS that gives an approximation of the action system is created, other LTSs are also created. These created LTSs just hide some information to reduce its size so that modal formulæ can be evaluated faster. The LTSs that are created are given in Table 5.2 and their purpose is:

1. Debug: Contains all actions; used for debugging purposes.
2. Node Port fire: Contains actions to indicate when a node fire as well as the input port that consume and the output port on which data is generated.
3. Node Fire: Contains actions to indicate when a node fires and no information regarding the ports of nodes.
4. CDFD input/output: Contains information of the input port of the CDFD that initiated execution, and the output port that generated data given a specific input port initiated execution.
5. Store access: Contains information on when nodes access data stores and if a read and write access violation occurs.

Table 5.1: Description of actions used in the ACP description that are also visible in the semantics of LTS.

Actions	Meaning
cdsa_input	Indicate which input port of the CDFD generated data
cdsa_output	Indicate which output port of the CDFD consumed data
cdsa_start	Indicate an input of the CDFD generated input data
done_fire	Indicate the node is done firing
env_start_done	Process environment done with starting nodes to fire
env_new_start	Process environment is ready for a new set of node to fire
error_r_w	Indicate a data store access violation
fire	Indicate a node fire
fire_ports	Indicating the input/output ports involved when a node fire
flow_action	A synchronised action to indicate that data is generated
flow_consume	A synchronised action to indicate that data is consumed
flow_query	A synchronised action to indicate that the flow containing data is being queried
node_select_in_port	Indicate node select input port that will consume data
selected_port_generate	Select an output port of a node that will generate data
start_fire	Synchronised message that indicate when a node fires
store_check	Synchronise the checking that data stores are accessed in a valid manner, node firing, and the “Environment” process.
store_update	Process node, to update access to data stores

5.6 Modal Logic

A mode logic [10] is defined by first defining a frame⁶ and then how the formulæ of the modal logical are evaluated. A frame consists of a set, S is state (also called possible worlds) and binary relation R between element of S , and denoted by $\langle S, R \rangle$. A formula can evaluate to true or false in a given state $s \in S$ and is recursively defined for modal formulæ P and Q , by

1. If the formula P is true in the state s , then $s \models P$.
2. $s \models P$ if and only if $s \not\models \neg P$.
3. $s \models (P \wedge Q)$ if and only if $s \models P$ and $s \models Q$.
4. $s \models \Box P$ if⁷ and only if for all $s' \in S$ such that $s R s'$, then⁸ it is the case that $s' \models P$.
5. $s \models \Diamond P$ if⁹ and only if there exist a $s' \in S$ such that $s R s'$ then it is the case that the formula if $s' \models P$ true.

⁶A frame is also called a Kripke structure.

⁷The operator \Box is also known as a “necessity operator”.

⁸The relation is defined for the pair of state s and s' .

⁹The operator \Diamond is also known as a “possibility operator”.

Actions	Debug model	Node Port Fire	Node Fire	CDFD input/output	Store access
cdsa_consume	X			X	
cdsa_input	X	X	X	X	X
cdsa_output	X	X	X	X	X
cdsa_start	X				
done_fire	X				
env_start_done	X				
env_new_start	X	X	X		X
error_r_w	X	X	X		X
fire	X	X	X		
fire_ports	X	X			X
flow_action	X				
flow_consume	X				
flow_query	X				
node_select_in_port	X				
selected_port_generate	X				
start_fire	X				
store_check	X				
store_update	X				X

Table 5.2: The visible actions in the different LTS models generated.

The LTS created in Section 5.5 is used as a Kripke structure by taking each state space as a state in the Kripke structure and creating a relation R_a for each action a in the LTS. The relation R_a is only defined for state s and s' (i.e., $s R s'$ is defined) if a transition exists in the LTS from the state space associated with s to the state space associated with s' , and is labelled by the action a . Thus creating multi-modal logic.¹⁰

5.6.1 μ -Calculus Formulæ

μ -Calculus [28] is a model logic where a modal “possibility” operator $\langle \text{action name} \rangle$ and “necessity” operator $[\text{action name}]$ are defined for each action. The LTS of Section 5.5 is used as the Kripke structure for the modal logic. This gives an elegant and powerful method to express properties of nodes that fire in a CDFD as well as determining if the properties hold.

The logic that μ -Calculus extends is the Hennessey-Milner logic with the following BNF grammar:

$$\phi := true \mid false \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \langle a \rangle \phi \mid [a] \phi$$

where $true$ is valid in any state and $false$ is never true in a state and a is a label. A formula is evaluated on a state of a Kripke structure where transitions are labelled, i.e., labelled by a 's. The formula $\langle a \rangle \phi$ is evaluated on a state by determining if there exist an outgoing transition labelled with a such that ϕ evaluates to $true$ on the state at the other end of the transition. The “necessity” operator is defined by $[a] \phi \iff \neg \langle a \rangle \neg\phi$, thus the formula $[a] \phi$ evaluates to true on a state s if the formula ϕ evaluates to true on all states connected to all transition with s as origin of the transition and labelled by a . Other connectives have their usual meaning.

The action labels used in the modal operator are extended to use a set of labels such that for a set A of labels it is the case that

$$\langle A \rangle \phi = \vee_{a \in A} \langle a \rangle \phi \qquad [A] \phi = \wedge_{a \in A} [a] \phi$$

where $true$ and $false$ are also added to the possible set labels with $true$ being the set of all labels and $false$ the empty set. Thus also allowing set complement $\langle \bar{A} \rangle \phi$ means that there exist an action $a \notin A$ such that it is the case $\langle a \rangle \phi$ evaluates to true. Regular expressions of labels are also supported:

$$R := \epsilon \mid a \mid R \cdot R \mid R + R \mid R^* \mid R^+$$

where ϵ is the empty sequence of actions (i.e., ϵ is the same as false), $[\epsilon] \phi = \langle \epsilon \rangle \phi = \phi$ and

$$\begin{aligned} \langle R_1 + R_2 \rangle \phi &= \langle R_1 \rangle \phi \vee \langle R_2 \rangle \phi & [R_1 + R_2] \phi &= [R_1] \phi \wedge [R_2] \phi \\ \langle R_1 \cdot R_2 \rangle \phi &= \langle R_1 \rangle \langle R_2 \rangle \phi & [R_1 \cdot R_2] \phi &= [R_1] [R_2] \phi \\ \langle R^* \rangle \phi &= \langle R \rangle \langle R^* \rangle \phi \vee \phi & \langle R^+ \rangle \phi &= \langle R \rangle (\langle R^+ \rangle \phi \vee \phi) . \end{aligned}$$

Fix-point operators can also be used to express mode properties. For an explanation of fix

¹⁰Usually a multi-model logic is also called a modal logic.

point operation refer to [28]. The only fix point operator used is the maximum operator ν and is used to indicate that an action occurs an infinite number of times.

5.6.2 Property Formulæ

Before a formula can be created to describe a property, a LTS must first be selected as this will determine the actions that are available for use in the formula. Therefore, select from Table 5.2 the LTS that will be used. Since a sequential execution model is used, a formula must express all possible ordering of nodes that execute simultaneously, when it is necessary to refer to nodes explicitly.

The formulæ used to verify the condition of an action are tested by:

1. There exist an execution path for each input port of the CDFD.

```
exists nid:NodeId.
  <true*.cdfd_input(IP0).true*.fire(nid)>true
```

Use the LTS in Table 5.2 with name “Node Fire”.

2. A node can consume data from more than one input port at a given time.

```
exists pid1,pid2:PortId.
  val(pid1 != pid2) &&
  exists pod1,pod2:PortId.
    <true*.env_new_start>
      (<(!env_new_start)*.
        fire_ports(Node_Cast, pid1, pod1)>true &&
        <(!env_new_start)*.
          fire_ports(Node_Cast, pid2, pod2)>true)
```

Use the LTS in Table 5.2 with name “Node Port Fire”.

3. Does an execution path exist starting from a specification input and result in the output port for the CDFD generating data multiple times.

```
exists cip:PortId.
  <true*.cdsa_input(IP0).true*.cdsa_output(cip)>true
```

Use the LTS in Table 5.2 with name “CDFD input/output”.

4. Determine if each node can fire at least once in any execution path.

```
exists pid:PortId.
  <true*.cdfd_input(pid).true*.fire(NICast)>true
```

Use the LTS in Table 5.2 with name “CDFD input/output”.

5. Determine if a node can execute an infinite number of times in any execution path.

```
nu X.<true*.fire(NICast)>X
```

Use the LTS in Table 5.2 with name “Node Port Fire”.

6. Determine if for all input ports of the CDFD and for all output ports there exist an execution path, starting with the input port and generating output on the output port.

```
<true*.cdfd_input(IP0).true*.cdfd_output(IP0)>true
```

Use the LTS in Table 5.2 with name “CDFD input/output”.

7. Determine if each execution path generated output

```
<true*.cdfd_input(IP0)>[cdfd_consume]
(exists cip:PortId.<cdfd_output(cip)>true)
```

Use the LTS in Table 5.2 with name “CDFD input/output”.

Formula 4 which is used to define redundant element in a CDFD is not strictly needed to determine if a design is valid, but provides information about a general property of a design.

5.7 Analysis using mCRL2

The analysis performed on the mCRL2 description also uses the tools that are available in the tool-set provided by mCRL2. The tools that are used to create the needed LTS and μ -Calculus evaluations are:

1. mcr122lps: Convert the translation in mCRL2 to a Linear Process Specification (LPS).
2. lpsparelm: Optimise an LPS by identifying elements that do not influence its behaviour, and remove those elements.
3. lpssumelm: Remove redundant summation from an LPS.
4. lpsconstelm: Remove parameters from processes that are constant from the LPS.
5. lps2lts: Convert from an LPS to an LTS.
6. ltsconvert: Use to convert the LTS to a simplified version using an equivalence relation.

The specification LPS consist of a single ACP processes in the form

$$X(d : D) = \sum_{i \in I} c_i(d) \longrightarrow \alpha_i(d) \cdot X(g_i(d))$$

where

- X is the name of the process taking a parameter d of type D .

- I is a finite index set.
- c_i is a predicate that indicates the corresponding multi-action is allowed in the current state.
- α_i is a multi-action parametrised by the current state.
- g_i is a state transform function that transforms the current state and corresponding multi-action that occurred.

Use of the LPS is an intermediate form necessary to create an LTS and evaluate the μ -Calculus formula. For evaluating μ -Calculus formulæ the utilities used are:

1. `lps2pbes`: Used to translate a μ -Calculus formulæ to generate a Parameterised Boolean Equation System (PBES).
2. `pbes2bool`: A PBES and a LTS is used to determine if the associated μ -Calculus formulæ evaluate to true.

Creation of μ -Calculus formulæ requires a LTS system where the tools use LPS and PBES files. A LTS and a LPS are seen as equivalent, since the two formats can be converted to each other. PBES is the method used to evaluate a μ -Calculus formulæ and no further consideration is given to these internal formats.

Other tools provided by mCRL2 which are useful when performing a manual analysis are `ltsgraph` and `lpsxsim`.

1. The tool `ltsgraph` is used to give a visual representation of a LTS if the number of state the LTS contains are not too large. Figure 5.5 gives the LTS of the small example (see Section 6.2) where the input and outputs of the CFD are of interest. Figure 5.6 shows how the LTS indicates nodes fire and how access to data stores are updated. A useful feature of `ltsgraph` is that initially only one state can be shown, and each state can be expanded and only then show outgoing transitions and target states. Such functionality is useful where the LTS is large, as is the case in Figure 5.6.
2. Using the tool `lpsxsim`, a list of allowed actions is given from which the next action is selected and so the user can step through all the different traces of a LTS, one at a time.

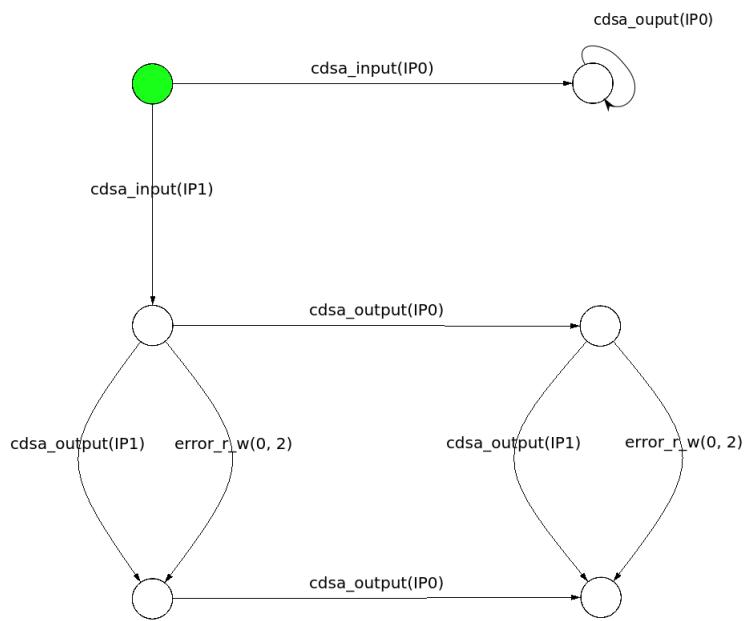


Figure 5.5: A visual representation of the small example's LTS that shows the input data consumed, output data generated and errors. Note: the loop link at the top right node was manually drawn.

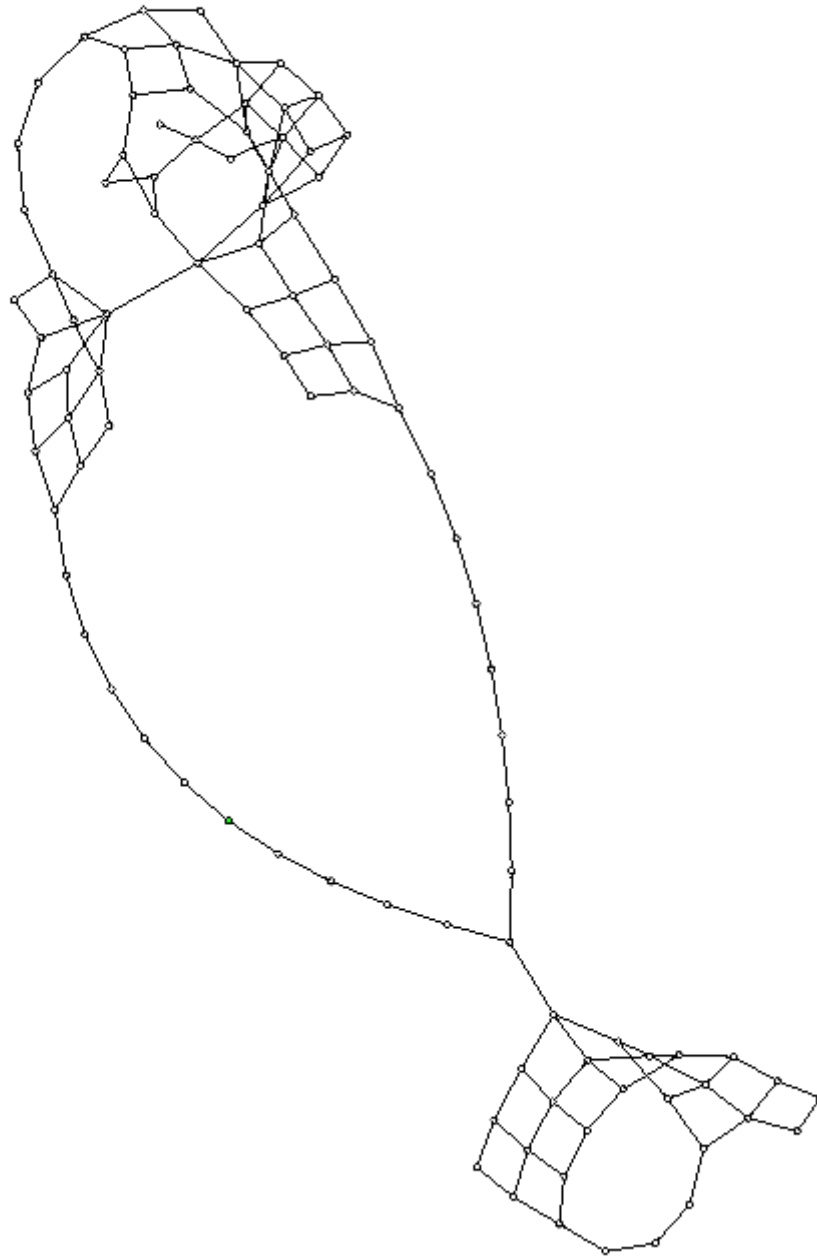


Figure 5.6: A visual representation of the small example's LTS that shows the actions that fire and data store access violations. All state labels are removed to give a better representation that shows the number of states.

Chapter 6

Developed Tool and Evaluation

For the created translation, a plug-in for Eclipse was developed (Section 6.1) which performs the translation automatically. This makes the creation of translations less error prone as well as efficient and repeatable by any developer.

Examples (Section 6.2) are used to illustrate the use of the tool and discuss its applicability. The first few examples show how specific properties of a design can be proven, specifically dynamic properties.

6.1 Developed Tool

Creating manual translations, as defined in Chapters 4 and 5, requires considerable work and is also error prone. This is addressed by creating an editor as an Eclipse plug-in to assist in the translation from a SOFL specification to both SMT-LIB and a process algebra description. Scripts are created using other tools to further analyse the translated files. The editor consists of the elements as shown in Figure 6.1.

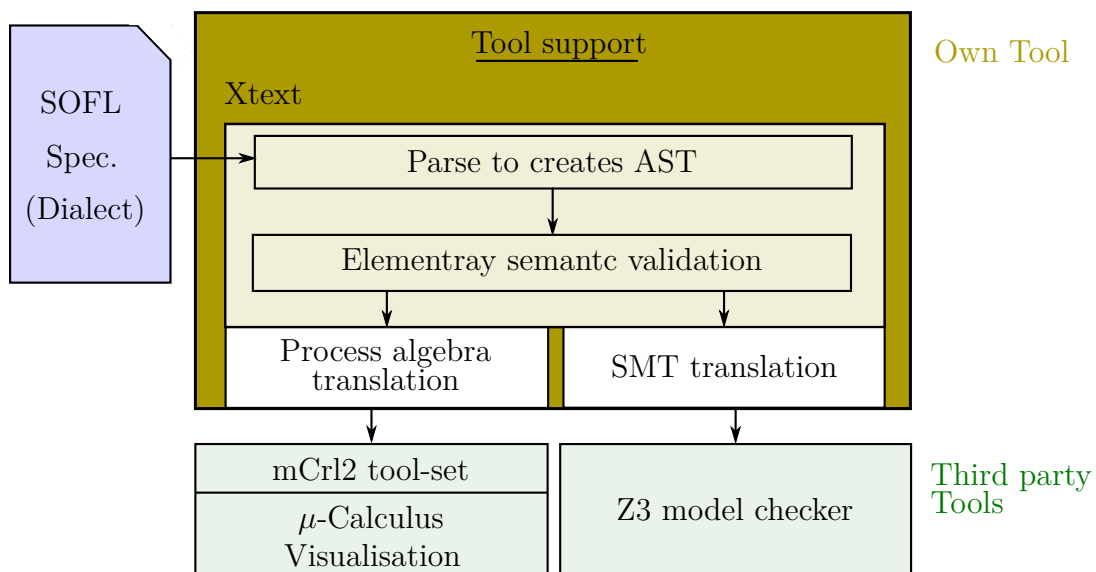


Figure 6.1: The elements of the created plug-in

Creating an Eclipse plug-in allows a wide range of libraries to be used such as the Xtext [6] library, which is used to create a Domain Specific Language (DSL) for our SOFL dialect, as well as creating an Eclipse plug-in. Most of the usability features of the tool are either provided by Xtext or are existing functionality in Eclipse.

A screen shot of the editor is given in Figure 6.2 which shows the workspace where a specification is entered. Section 6.1.3 gives details of how the semantics are verified using Xtext. Analysis of the created translation is done in a terminal with the results also displayed in the terminal. A markdown report of the formulæ used to verify the static semantics is also created, but the report only has value in identifying the predicate that was involved in creating the specific formulæ. The versions of the components used by the tool are:

Eclipse 2018-12

Xtext 2.17

Z3 4.8.1 - 64 bit

mCRL2 201808.0.e419836342M

and the tool is available at <https://gitlab.com/JohanVdBerg/sofl-editor.git>.

6.1.1 Workspace

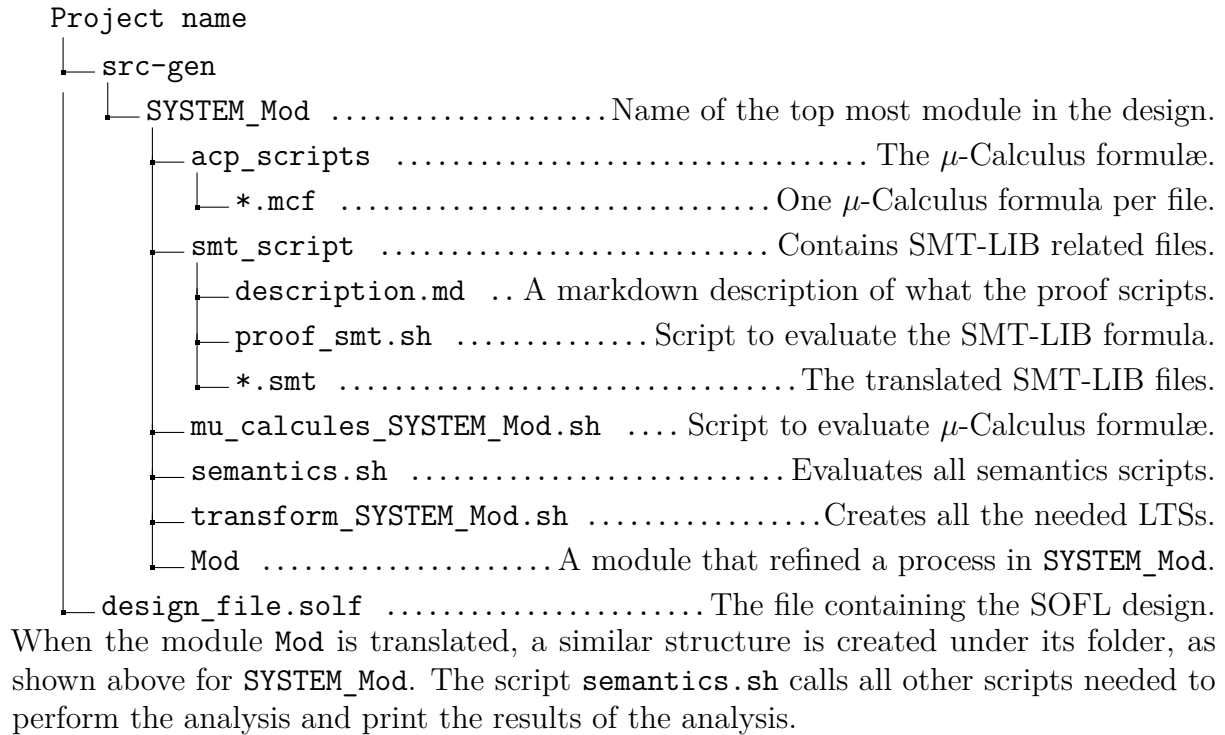
The workspace (Figure 6.2) consist of three areas:

Project explore: Multiple projects can be open, and there is no restriction on the number of files that define SOFL modules in each project. For each project a separate scope exists which shows the elements already defined and displays them in a pop-up menu for ease of use.

Editor area: A text editor area used to enter a dialect specification. Xtext parses a specification file using the ANTLR framework [65] and creates an Eclipse Modelling Framework (EMF) model of the specification. A framework is provided by Xtext that calls functions during and after the EMF model creation. These functions are used to validate a design and create the needed translations.

Specification outline: The outline gives an overview of the active SOFL module in the editor area.

The first step to create a SOFL specification is to create a project where all specification is created. Subsequently, the translation to SMT-LIB and mCRL2 are created when a specification file is saved and passed validation. A similar directory/file structure of the project is created when the translations are created, e.g., a top module `SYSTEM_Mod` where the module `Mod` refines a process in `SYSTEM_Mod` is given by:



More than one top module (specifying more than one system) can be described under a project, however scoping will not indicate which element is from which specification. When a module is translated, its module hierarchical structure is used to determine the location where the translated files are created. Also, more than one model can be defined in a file.

6.1.2 Xtext Parsing

An implementation of the dialect’s grammar (see Appendix A) is created using Xtext specific syntax text. The language of Xtext is based on Extended Backus-Naur (EBNF) which allows the definition of a reference element. An element that can be referenced is defined by a parsing rule of Xtext. This referencing information includes some semantic information in the Xtext description, but it is useful in understanding the grammar as the reference element provide additional context of where the elements are used. When a parsing rule is used as a referenced element it must have a name (specified as a string) that is used to refer to the element.

After the grammar of the dialect is defined, it is translated into a format that the ANTLR parser framework accepts as a definition of a parser. When a specification in the dialect is processed, it is first parsed by the ANTLR parser. After successful parsing, Xtext performs some semantic validation and translation of the specification.

6.1.3 Xtext Semantics Verification

Xtext provides a number of “call-backs” that are used to perform partial static semantic verification of a specification in the dialect. One call-back is used to define the scope used when elements are referenced and other “call-backs” are used to define custom verification. These checks are integrated with Eclipse so that checking “continues” as the user types

the specification. Scope checking gives the user suggestions as to which elements can be referenced. Verification of semantics is also performed. When the semantics is not satisfied, an approximation of the place where the elements failed the verification is underlined. The verifications performed by Xtext are:

1. All element referenced are in the correct scope.
2. The element used are of the correct type.
3. Data flows are correctly connected.
4. CDFDs introduce new port and data flows correctly.
5. External data stores accessed by either `ext rd` or `ext wr` can only be externally referenced by a module if they are visible in the process being refined.
6. When module have a parent define then the CDFDs in the module refine a process that is defined in the parent.
7. Name of processes start with a capital letter, and no duplicated process names are allowed.
8. Name of the top most module (a module without a parent) starts with `SYSTEM_`, the name of all other module structures begin with a capital letter.

When all these checks pass, an EMF model of the parsed data is created by Xtext that satisfies sufficient conditions so that the specification can be translated.

6.1.4 Translation Generation

When a specification is saved in the editor and all the checks performed by Xtext are passed, the EMF model is translated to SMT-LIB and mCRL2 and a markdown document is created to describe the SMT-LIB formulæ created.

6.1.5 Final Analysis Step

The file `semantics.sh` needs to be run so that the static and dynamic verification is performed. The designer needs to manually verify the output of the analysis. For example, not all output ports of a CDFD generate data for every input port that resulted in its “execution”. This might be valid for the system being designed but the designer needs to make the decision.

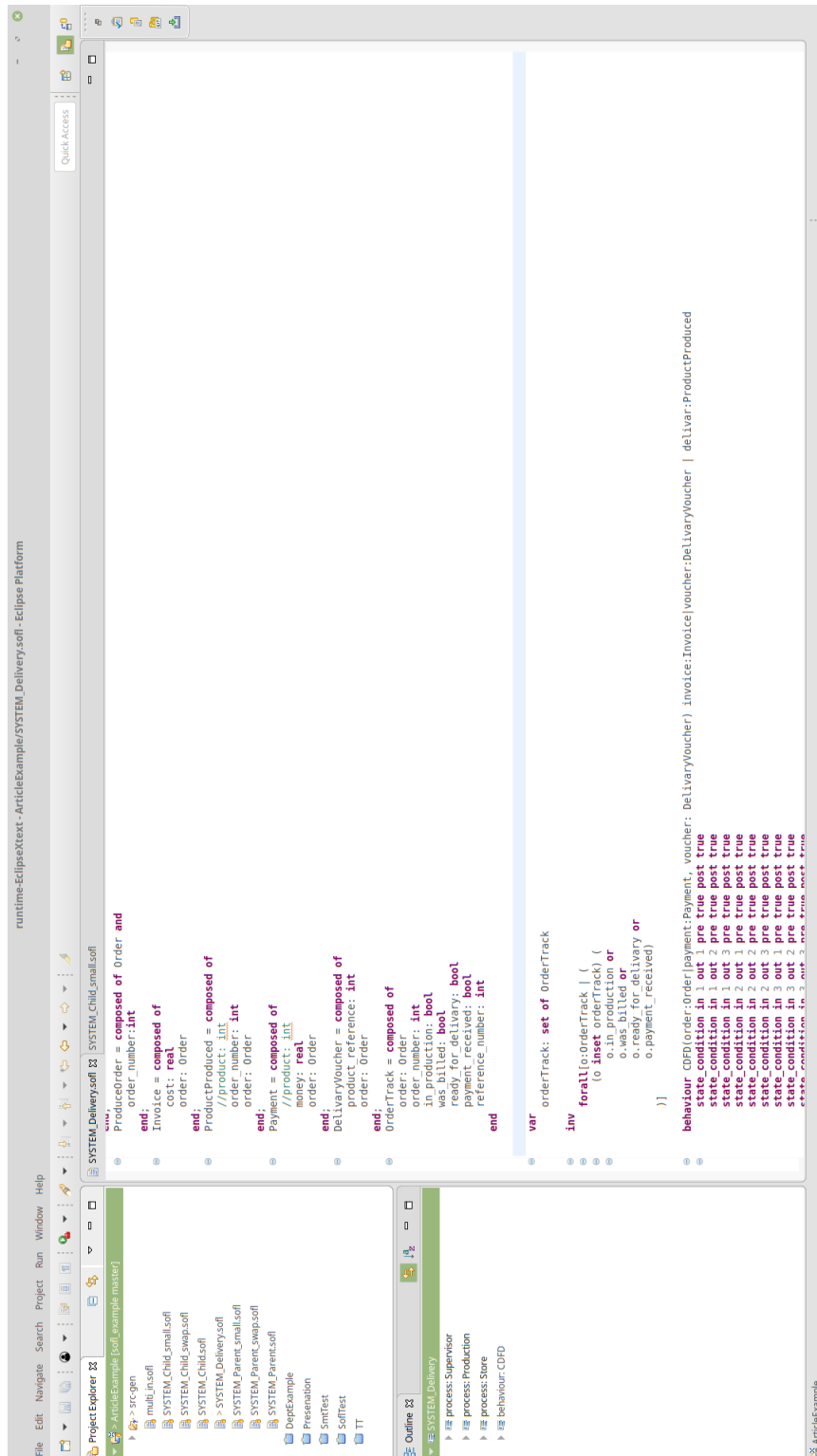


Figure 6.2: Editor for the syntactic and semantic analysis of SOFL specifications

6.2 Examples

Three examples are considered in this section; they are created with the purpose of verifying properties of a design and checking that translations from the specification and the tool, function correctly.

6.2.1 Small Example

This example is used to illustrate a number of dynamic properties:

1. when a process can execute an infinite number of times
2. determine the output port of the CDFD that can generate data given that a specific input port initiate the execution
3. identify when a process cannot fire in any time

and determine how the static verification performs:

1. verification of each pre and post condition
2. illustrate limitations in the generation of the proof obligations

The CDFD analysed is given in Figure 6.3. The specification is given in Listings B.1 and B.2, the translation to mCRL2 in Listing B.3, and the results in Tables B.1 and B.2.

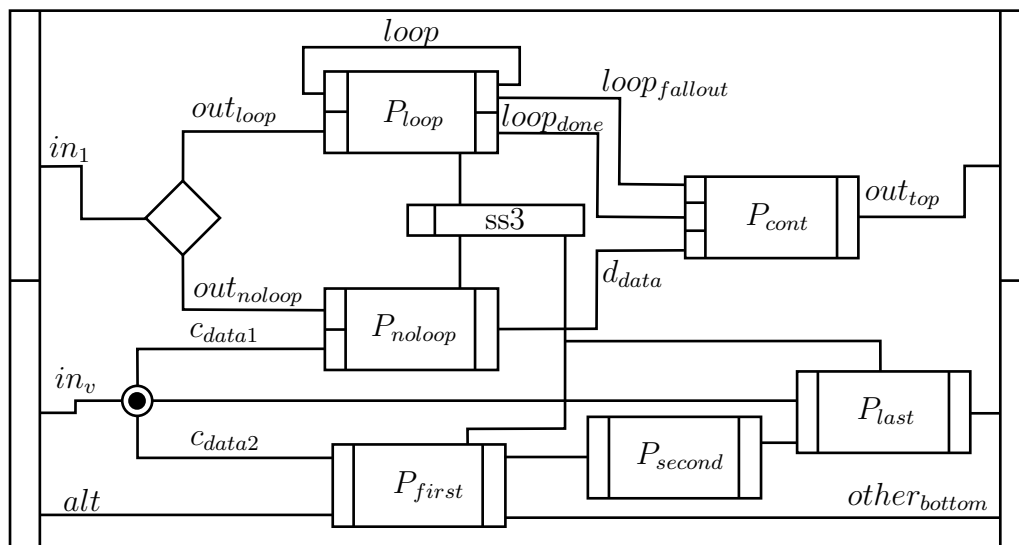


Figure 6.3: CDFD of a module in the small example that refine the process ToRefine

The dynamic properties were identified as expected and a summary of the results are given by:

1. Every node can execute at least once.
2. Processes P_{loop} and P_{cont} can execute an infinite number of times.

3. The bottom output port of the CDFD cannot generate data if the top input port initiates “execution”.
4. Each input port of the CDFD that initiates “execution” can generate data multiple times on an output.
5. Read/write access violations to the data store exist, and the violating processes are given.

No indication (except for where there are read/write violations) is given on why the result of the properties was obtained. The designer needs to perform a manual analysis. For example: The reason that the lower input port of the CDFD can generate output on both output ports is due to the broadcast node that send data to both processes P_{noloop} and P_{first} .

The verification performed by Xtext was satisfactory, but improvements will allow better integration into Eclipse and give the user/designer better support. For example, the list of suggested names of the identifier can be shortened by using more type and location information in the specification, where the suggestion is made.

The verification performed by the SMT-Solver (Z3) was the most challenging. Whenever a SMT-LIB formula failed, possible reasons for such failure included:

1. An error in the SOFL specification.
2. An incomplete SOFL specification.
3. The predicate created by the translation is not provable by an SMT-Solver. This option was not further analysed.
4. Limitation of the translation into SMT-LIB.

Determining the reason for the failure of a formula will require work from the designer. The first two cases of failure are part of the normal design process of a system and the tool provided satisfiable, not satisfiable or unknown result.

The last case of failure is due to the translation to SMT-LIB which is defined in Chapter 4 and the limitations are highlighted by the example:

Limited “history” In the example there are two cases P_{loop} and P_{last} . The process P_{loop} has a feedback that results in the process being called a number of times. The number of iteration is dependent on the data value that flows back and the predicate, which would require both static and dynamic analysis.

SMT-LIB An SMT-Solver effectively¹ searches for an assignment for the free variables in a formula, and when such an assignment is found the formula is satisfiable. This property make it possible to create a translation that must be true but cannot be solved, consider the translation:

```
(declare-const out_v Int)
(assert (forall ((t_loop_1 Int))
```

¹Note, techniques do exist that determine if a formula is satisfiable without determining the assessment.

```
(=>
 (> t_loop_1 20)
 (and (> out_v 30)(> out_v t_loop_1))))
```

There always exists a variable `out_v` such that `(> out_v t_loop_1)`, but no assessment can be defined for all cases, and therefore the formula cannot be proven satisfiable by a SMT-Solver. One possible solution for such a case is to use:

```
(assert (forall ((t_loop_1 Int))
 (=>
 (> t_loop_1 20)
 (exists ((out_v Int))
 (and (> out_v 30)(> out_v t_loop_1))))))
```

This formula does not have any free variable; therefore, the problem of finding an assignment does not exist any more. Unfortunately, computation performance for such formulæ was very poor, and so the use of this technique was avoided. Since the internal working of a SMT-Solver was not investigated, no conclusion can be made about whether a theorem prover would perform better than a SMT-Solver or whether just a better translation is needed.

A number of read/write access violations exist, where the output describing one of them is given by:

```
cdsa_input(IP1)
env_new_start
start_fire(Node_P_noloop)
start_fire(Node_P_first)
store_update(ss3, write_access_begin)
store_update(ss3, write_access_begin)
error_r_w(0, 2)
```

where `ss3` is the data store accessed with write access twice, the top port initiate “execution” of the CDFD, and the two processes involved are P_{noloop} and P_{first} .

6.2.2 Large Example

A larger example that is a three times duplication of the CDFD of the small example, is analysed here. The example was created when the computational limitations of the process algebra was being determined. The CDFD of the large example is shown in Figure 6.4.

The result of the dynamic analysis could not be completed, since the process of conversion from the mCRL2 description to LTS did not terminate, and the PC used “froze”, since the number of traces that resulted from the sequential model of execution was too large.

Consider a set of node that need to fire next which contain n nodes. The number of traces that is created during translation is n factorial and it appears that this is the

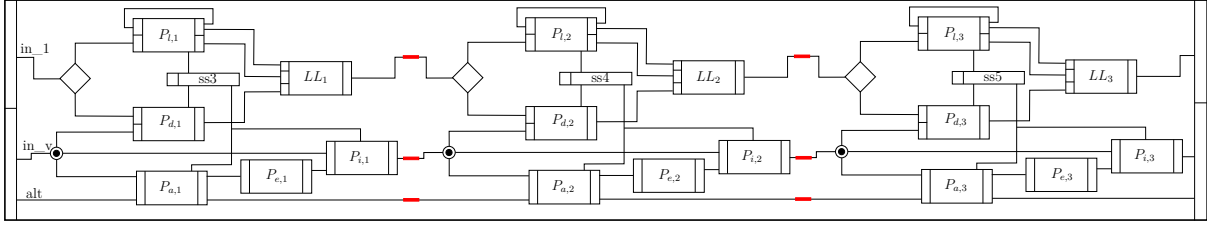


Figure 6.4: CDFD of the large example that duplicates the CDFD of the small example three times

reason why the conversion process did not terminate: The example consists of 24 nodes, if only half of them need to fire simultaneously, the number of different traces that need to be generated are of the order 2^{79} . If each trace requires one byte of data, 2^{73} Gigabyte of data is needed.

By swapping the input port to which `in_1` and `in_v` are connected, an LTS could be created, since this modification effectively disables the bottom half of the process in the CDFD, i.e., $P_{a,j}$, $P_{d,j}$, $P_{e,j}$ and $P_{i,j}$ for $1 \leq j \leq 3$. Thus 15 nodes are disabled, leaving 9 nodes which gives an upper bound 2^{19} for the number of traces that need to be generated. This shows the sensitivity of the dynamic analysis's success based on the structure of the action system created.

6.2.3 Multi-port Example

The purpose of this example is to create a situation where two ports of a process are allowed to fire simultaneously for the purpose of verifying the μ -Calculus formula that determines if a node can consume from more than one input port at any given time. The specification is given in Listing B.4.

Analysis of the multi-port example did show that a node can consume data from more than one port. By connecting both input data flows to the same input port the process cannot consume data from more than one input port any more.

6.2.4 Producer Delivery Example

A different approach to creating a formal design based on data flow diagrams was considered in [11], where the main difference of their approach is that a data flow diagram is seen as processing stream process. The system described in [11] is used there to determine how good the system can be described in the dialect and how the developed tool can analyse.

A production and delivery system is shown in Figure 6.5, and a specification in the dialect is given in Appendix B. The figure shows a different type² of data flows diagram, the meaning of the data flows are:

Delivery enable: The items are ready for the client to receive them.

Invoice: A client is invoiced for an order placed.

Item: A client received the items ordered.

²Each process in the diagram contains a state that is persistent.

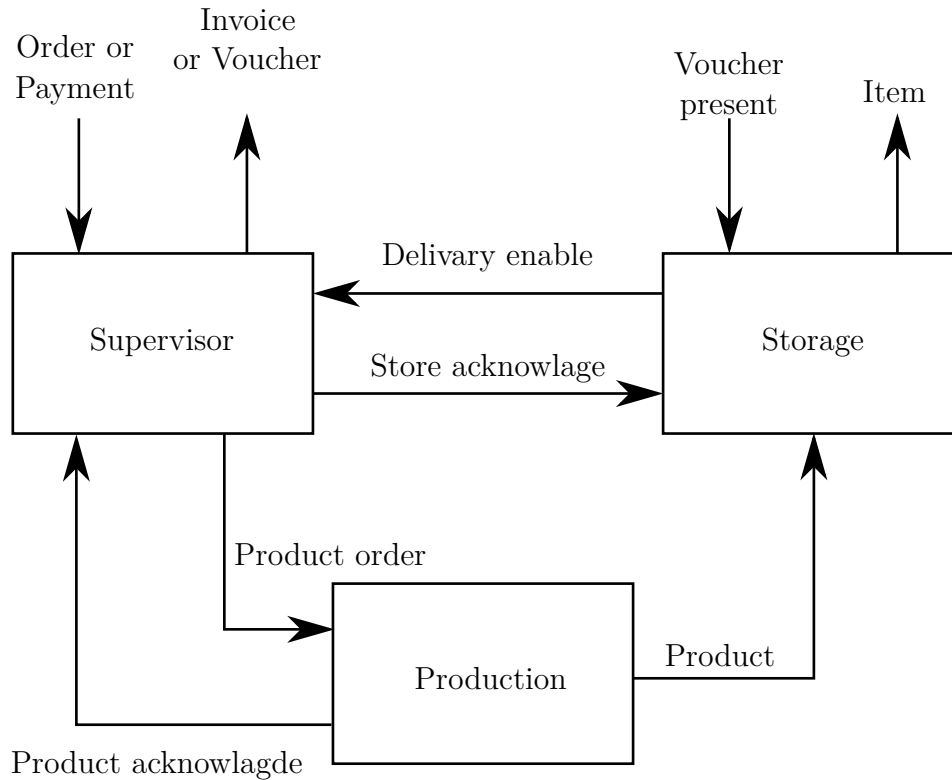


Figure 6.5: State transition diagram of a delivery system

Order: A client placed an order.

Payment: A client makes a payment for an order.

Product: The items produced are sent to storage.

Product acknowledge: The supervisor is informed of the items produced.

Product order: Production of items starts.

Store acknowledge: The store is informed that a voucher was given to a client.

Voucher: A client receives a voucher for goods produced.

Voucher present: A client gives a voucher to receive the produced items.

The actors in the system are as follows:

Client: The client orders a product, and the process requires the following steps: place order, pay for the order, receive a voucher, and use voucher to receive product.

Supervisor: The supervisor interacts with the client and the rest of the system so that the correct business processes are followed. A supervisor allows production to start, generates invoice, receives and acknowledges payment.

Production: Production has the following steps: produce product, store product, and inform supervisor product was produced.

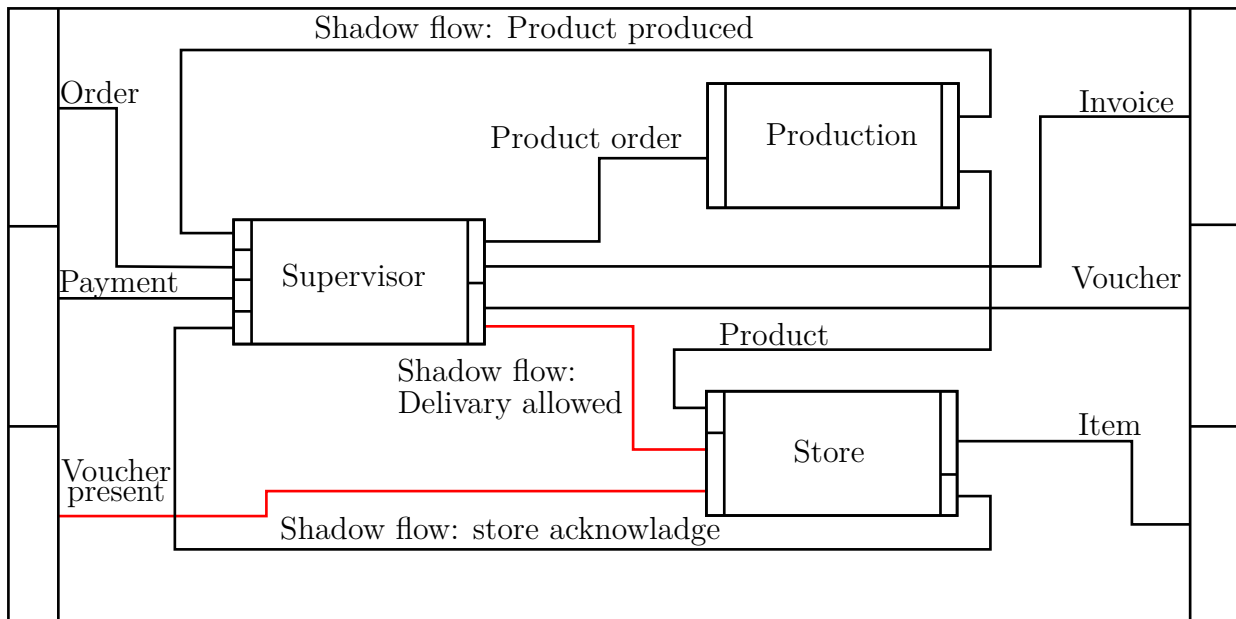


Figure 6.6: CDFD of producer example

Storage: Products are stored after production until the client accepts delivery of the product.

The CDFD of its specification is given in Figure 6.6. Each actor also consists of a state, e.g., a client will only attempt to receive a product after delivery was not received. The first challenge is the handling of this state information in both the dialect and the SOFL. Only the state information of the supervisor is approximated by

```
var orderTrack: set of OrderTrack
```

and the problem of persistence between interaction of the client with the system is ignored.

Verification of SMT-LIB formulæ could not be performed³ since data types besides numerical types were used. This highlights that more complex data types are needed in the translational to SMT-LIB. However, it does not necessarily mean that types like classes should be included in the dialect. Composite data types also include “inheritance” of members for example:

```
type
  Order = composed of
    product_type: int
    amount: int
  end;
  ProduceOrder = composed of Order and
    order_number: int
  end;
```

³The tool fails this phase by only indicating a syntax error in the generated SMT-LIB files. Dynamic analyses are not influenced by these errors.

An alternative to classes is to use composite type and functions and therefore obtain elementary⁴ “class-like” behaviour in a procedural programming language.

Results from the dynamic analysis did reveal the following:

1. The `voucher` input port could not generate output and results in the creation of an execution path: This is due to the modelling done in the dialect and is true to SOFL. To resolve this the fire rules of the nodes must also consider the state of the data store `orderTrack`, since it contains information about whether the product was produced or not.
2. The μ -Calculus formulæ involved in determining if process could fire an infinite number of times did not terminate: This shows the sensitivity of the dynamic analysis’s success is based on the structure of the action system created. When looking at the CDFD of the small example it was expected that these formulæ could have terminated.
3. The μ -Calculus formulæ involved in determining if process could consume data from more than one input port did not terminate: A better translation will determine if a node can consume data from more than one input during generation of an LTS, similar to the verification of read/write access violations.

Properties that describe the behaviour of interactions between the supervisor and client could not be modelled, even using a custom μ -Calculus formula. This limitation is due to the abstraction of data values by tokens which meant that data values could not be used during the creation of formulæ.

Only properties that look into the “future” can be described using μ -Calculus. For example, it was not possible to say: Whenever a product is produced an order must have been placed previously. This did not limit the ability to create formulæ to verify that the design is valid, since the direction that data flows through a data flow diagram is the same direction that an element can be referenced in a μ -Calculus formula.

6.3 Time Analysis

Time measurements were done on an Intel[®] Core[™] i7-6700 CPU with 32GB of memory. Each measurement was taken only once.

Timing results

For the small example, the time needed to analyse the translation of the module refining a process is given by:

1. create all the needed LTS systems: done in 20 seconds
2. prove all the predefined μ -Calculus formulæ: done in 49 seconds
3. prove all the SMT-LIB formulæ: done in less than a second

⁴For example, polymorphism is not possible

This gives a very optimistic view of performance, as each CDFD is analysed separately.

Analysing the large example, the times of the corresponding module as analysed in the small example are given by:

1. process to create the first LTS was stopped: PC performing the computations ran out of memory, 32 GB.
2. prove all the predefined μ -Calculus could not be done due to lack of LTS.
3. prove all the SMT-LIB formulæ: done in 1.5 seconds.

The result of the larger example does not give a positive outlook for the created tool. The considerable increase in resources needed to create a LTS is further discussed in Section 5.7. By swapping the input port to which `in_1` and `in_v` are connected, the time needed to create a LTS is reduced to 2 minutes.

Analysing the producer example, the times of the corresponding module as analysed in the small example are given by:

1. The process to create the first LTS: done in 9 seconds.
2. Prove some predefined μ -Calculus: done in 24 seconds. The formulæ that test if a process can execute an infinite number of times and consume from multiple input ports did not terminate in a reasonable⁵ time.
3. Prove SMT-LIB formulæ: not done due to limited translation.

Data collection for estimating the time needed to perform analysis was not very successful, since the small CDFD already resulted in the analysis not terminating, and the few times measured were very erratic. The reason that the measure times were erratic is due to the complexity of predicating the work load needed for the analysis performed by Z3 as well as mCRL2.

Discussion SMT Timing

The amount of work needed to determine if a design's static semantics is satisfied is dependent on:

The amount of nodes Each node define a set of equations to be solved, let N be the set of nodes.

Size of nodes For each port pair a set of equations are defined that need to be solved. For a node with n input and output port, n^2 such sets need to be defined.

Connectivity Data flows between nodes determine which predicates are related, specifically the pre-image of a port is determined by the data flows. For a node with n input port where the maximum number of ports connected to an input port is given by m , the amount of work is directly related to $n \cdot m$.

⁵This was 4 hours. Compared to 24 seconds for all the other formulæ, it is far too long.

CDFD Additional ports introduced by a CDFD do not determine the number of formulæ that need to be solved. The determining factor is the number of input and output ports of the process being refined, and let n be the number of input and output ports in the process. The number of formulæ created is given by $2 \cdot n$.

The complexity will only be expressed by an approximation of the number of formulæ being solved. This was done since determining (if possible) the amount of work needed to solve formulæ is not easy. Thus, the amount of work is estimated by

$$\sum_{i \in N} (n_i^2 + n_i \cdot m_i) + 2 \cdot n_c + 1$$

with N the set of node on the current CDFD and n_c the number of input/output port in the process refined by the CDFD. An upper bound is created for the estimation by

1. Use the maximum number of input or output ports that a node contain.
2. The process being refined is also contained in a CDFD and the same estimation is used for its number of port as in the current CDFD.
3. The largest size of a port's pre-image is used instead of considering the pre-image of each input port of a node.

Let n be the upper bound for the number of ports and m for the connectivity, which results in

$$\begin{aligned} & |N| (n^2 + n \cdot m) + 2 \cdot n \\ & \leq |N| (n^2 + n \cdot m) \end{aligned}$$

Thus the designer needs to use only a small number of nodes in a CDFD, and the nodes must not be highly connected to other nodes. Alternatively stated, the width of each level in a refinement hierarchical structure must only increase moderately. The connectivity between node is determined by how much functionality is contained in each process. Grouping functionality creates an abstraction that needs to be refined until each process is "concrete" enough to be implementable. Thus, by decreasing the connectivity of the nodes the depth of the refinement hierarchical structure and the number of nodes in a CDFD increases.

Luckily, these guidelines for creating a design (that does not require too much work to determine if its static semantics is satisfied) are in line with "good" design principles.

1. Group elements in a design with similar functionality in the same CDFD.
2. Consider only the relation between different computational element that are on the same "level". For example, do not mix data flows related to used input with data flows related to system internal information in the same CDFD whenever it is not strictly needed.

It is not recommended that "time needed to determine if static semantics is satisfied" be used to determine if "good" design principles were used. Rather, improvements in the

time needed to prove that the static semantic are satisfied, are a possible advantage of creating a “good” design where the elements are sufficiently grouped.

Luckily, the complexity is still polynomial-time.

Discussion on Process Algebra Timing

The example showed that the amount of time needed to create LTSs and evaluate the μ -Calculus formulæ is very erratic. The only property that could be identified that will give better performance is to create a better translation into mCRL2. One solution is to define an ordering for the nodes and only create one trace for a set of nodes that fires. This will allow considerably larger CDFD to be analysed.

6.4 Correctness of Tool

The example in this section was used to test the developed plug-in. This approach is comparable to an SBT; however, the number of inputs used for verification here is negligible compared to the number of possible specifications. The author is not aware of any other approach to automatically generate specifications so that the generated specification can be used to verify the tool.

Chapter 7

Future Work and Discussion

This chapter contains suggestions related to where the developed tool, and how the semantics, can be improved. A comparison is also made with the semantics defined for SOFL in [46] in order to determine if the dialect is completely different from SOFL.

7.1 User Experience

Formal methods has a reputation for not being user-friendly. The work done here has improved the user experience, however, there are still opportunities for improvement. Considerable work is still required during the analysis of the results to determine the reason why aSMT-LIB formulæ failed. This can be addressed by improving the editor in order to indicate which predicates were involved in creating a formula that failed.

Custom descriptions of dynamic properties will considerably improve the usability of the dialect without the designer having to write μ -Calculus formulæ. For example, whenever a customer places an order for n items, the producer will be informed $\lceil n/2 \rceil$ times, and produce $2\lceil n/2 \rceil$ items.

Research in SMT-Solvers improved when a common file format¹ was created. One solution will be to create an open-source editor for SOFL so that all research can be integrated in the editor.

7.2 Language usability

For the purpose of defining a formal semantics the dialect created is a simplification of SOFL. The expressibility of the dialect was not considered and neither was how well practical problems can be described by the dialect.

7.3 Correctness of Tool

Not much was done here to determine if the developed plug-in is correct, but usage of the Xtext library did help considerably. The library is well supported and easy to use with a

¹SMT-LIB

lot of functionality provided “free”. The library Xsemantics² can be used to improve the type checking and it integrates with Xtext which will help to reduce errors. Both libraries also use syntax that is very close to the syntax used by theoretical work. This helped in creating the tool without requiring too much translation. Use of a tool like “OTT” [67] will also assist in defining an operation³ semantics and in improving confidence in the semantics’ correctness.

7.4 Specifying Properties

The μ -Calculus formulæ created can only reference a node that fire when data is consumed or data is generated. For data values to be used in the formulæ, predicates in the dialect must be implemented using the rewrite system of mCRL2. It is not clear if it would be more efficient to design a new process algebra that has a stronger inference system in which the predicates can be expressed, or use an existing process algebra.

Also, it not possible to make reference to the “past”, which would require a model logic with a “future” and “past” operator. This will help considerably to improve the number of properties that can be proven by a model logic.

Currently, neither the dynamic nor the static semantics can prove all the properties of interest in a loop. The process P_{loop} in the small example is an example where termination cannot be proven, and only the possibility of infinite firing is proven by a modal formula.

7.5 Semantics

Condition 15 requires that all input and output ports of a CDFD must be connected by an “execution” path. This requirement is not always reasonable. An improvement will be that: each input port of a CDFD must be connected to at least one output port with an “execution” path. Using the improved condition, it will also be useful to use refinement for testing if all the necessary execution paths do exist and not just verify that an execution path exists for each input port of the process being refined, i.e., also use a process algebra description in the verification.

7.6 Thoughts on SOFL

SOFL follows an approach that focuses much more on the designer than the approach taken in this dissertation. By contrast, the approach used in this dissertation is to create a semantics that will allow properties of a design to be proven. This section discusses the work done in this dissertation and compares it to the description of SOFL as given in [46].

7.6.1 Non-determinism

Non-deterministic broadcast directs its output data to a data flows connected to a node that will have the best chance of firing, when the next set of nodes fire. Therefore, a

²<https://github.com/eclipse/xsemantics>

³Behaviour

non-deterministic broadcast node must be able to “see the future”.

This property of being able to “see the future” creates difficulty in the semantics of the dialect, as the next state is only dependent on the current state and the set of actions selected to fire. Thus, defining the semantics would require modal logic that can be used to describe the current state and “future” states and thereby increase the complexity of the semantics. This requirement of a modal logic would have an impact on the translation used as follows:

1. The translation to SMT-LIB would need to be updated: The pre-condition of all nodes connected to the non-deterministic broadcast node would define a state space that the output of the non-deterministic broadcast node must satisfy.
2. Radical changes will be needed in the translation to mCRL2 as the “see into the future” property would need to implement a non-deterministic broadcast node to determine which of its connected node will have the best chance to fire during the next set.

Larger computational penalties are expected during analysis of a translation to mCRL2 when a non-deterministic broadcast node is included in the dialect.

7.6.2 Classes and Data Types

Before classes can be included, a semantics for classes needs to be defined and the similarity between classes, and module must be addressed. An alternative will be to allow the user to define custom ADT or extend SOFL to include more predefined types.

7.6.3 Semantics

The semantic given in [46] is not formally defined which makes it difficult to give comparison between the difference in semantics of SOFL and the dialect. In this section, a discussion is given of some of the differences.

The book [46] has a different approach to define the semantic of SOFL, where the semantics defined in this dissertation follows the approach:

1. First allow a larger number of specification to have a “mathematical model”.
2. The larger number of allowed models are required to verify the needed properties.
3. Conditions given in Section 3.2 are tested to determine if a model created by a design is valid, and therefore also if a design is valid.

These proprieties are verified using either Xtext, Z3 or mCRL2, where the specific tool also determines the limitation of the verification performed.

Xtext parsing and validation are used to test if data flows correctly connect ports in a specification, define the scope of variables, and determine if the variables are used at the correct places, to name just three checks. These verifications are related to [46, Def 15 and 27].

Node and invariant verification is done by static semantics verification:

The dialect requires that a node is realisable and this is equivalent to [46, Def 24 and 29].

Each node is required to fire only in a valid state. In the dialect, this is verified by creating an approximation of execution paths that can be verified during static analysis. Definition [46, Def 30] also requires that pre and post condition of an “execution” are all valid, but no clear method is given of how the verification is done for the semantics of SOFL.

The invariant verification is stricter here since [46, Def 28] only considers one variable of the invariant at a time. The method given here considers all the variables of the invariant and all are tested simultaneously.

The influence of initialisation is defined in [46] but is left to the designer to include where needed. Here, tests that cover the influence of the initialisation conditions are defined.

Use of FOL allows a translation to SMT-LIB without the needed complexity of working with LPF. More importantly the “hidden” relation between input (or output) ports that can be created using LPF is not possible in the dialect, and allows a “cleaner” separation between static and dynamic semantics.

Refinement verification is done by static semantics verification:

The idea that the output state space of a CDFD must be contained by the output state space of the process being refined, is similar to the conditions required by [46, Def 17]. How the refinement are verified is discussed more explicitly in this dissertation.

Two definitions are given in [46] which define two different approaches to verifying refinement:

1. In [46, Def 22] the conditions

$$\begin{aligned} pre(a) &\longrightarrow pre(b) \\ pre(a) \wedge post(b) &\longrightarrow post(a) \end{aligned}$$

where processes b is a refinement of process a , need to be true for the refinement to be valid. In the dissertation, the last condition is replaced by $post(b) \longrightarrow post(a)$, and this creates a bigger separation between the refined process and process being refined. Thus, it does not include any predicate transform-like information in the refinement process. This is also consistent with the view that a predicate cannot define a state change, only a CDFD and a node with its pre and post conditions, can define state change. But as an after thought the information of the pre condition ($pre(a)$) should have been included, here.

2. Definition [46, Def 23] requires the refined process to be defined by an explicit specification. Since explicit specification are not considered in the dialect, this definition is not applicable. It is still worthwhile mentioning that [46, Def 23] is stricter in its requirements than [46, Def 22] and requires an evaluation of the

refined “function” for each of its allowed inputs, thus using dynamic semantics during the verification of static semantics.

CDFD The approach taken in [46] requires more definitions, such as defining a starting and terminating node. In addition [46] relates those elements with each other to define the properties that a CDFD must satisfy, thus adding an additional layer of abstraction.

1. Termination of a CDFD as defined in [46, Def 10] is the same condition used for the dialect, and termination is tested by verifying that there are no nodes that can fire an infinite number of times.
2. The execution path available in the created action system are equivalent to the data flow groupings of [46, Def 16].

This can easily be rectified by not spreading the related definitions over several pages in the book and “optimising” the definitions.

Shadow data flows are also introduced in this dissertation to avoid empty ports. Shadow data flows are a possible alternative to passive⁴ data flows, as they do not require the need to introduce an extra mechanism to define the semantics of passive data flows. Also, shadow data flows fit in better with the idea of data flow diagrams.

7.7 Conclusion

This dissertation has discussed the creation of a dialect of SOFL, as well as defining its semantics by translation into SMT-LIB and mCRL2. A plug-in was also developed for Eclipse to create the translation from a specification in the dialect. Finally, examples were used to identify the properties that can be proven and the time required to obtain results.

The dialect is not a replacement for SOFL since the dialect does not include an equivalent of non-deterministic broadcast structures. Before the dialect can be considered as a replacement, it first needs to be shown that at least the system types can be specified and that it contains no equivalent element to a non-deterministic broadcast structure.

The developed tool can be improved in two places: better support for data types in the SMT-LIB translation, and use of a translation to mCRL2 that is not so resource “hungry”.

The approach taken allowed the removal of LPF and therefore removed the possibility of a “hidden” relationship between input ports of a node. The use of modal formulæ to describe dynamic properties of a design does provide a considerable advantage despite the resource “hungry” translation that is currently implemented.

The dialect puts a lot of focus on the idea that a data flow diagram consumes data as soon as possible, and at the end, there is a suspicion that it may be the source of some of the dialect’s limitations.

⁴Passive data flows were not included in the dialect

Bibliography

- [1] Back, R.J.R., Kurki-Suonio, R.: Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **10**(4), 513–554 (1988)
- [2] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. LNCS, vol. 6806, pp. 171–177. Springer (2011)
- [3] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., University of Iowa (2017), <https://www.SMT-LIB.org>
- [4] van der Berg, J., Gruner, S.: Formal semantics and tool support for a syntactically restricted dialect of SOFL. In: *Structured Object-Oriented Formal Language and Method*. LNCS, vol. 11392, pp. 125–145. Springer (2019). doi:10.1007/978-3-030-13651-2_8
- [5] Bergstra, J., Klop, J.: Process algebra for synchronous communication. *Information and Control* **60**(1/3), 109–137 (1984). doi:10.1016/S0019-9958(84)80025-X
- [6] Bettini, L.: *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd (2016)
- [7] Blackburn, P., De Rijke, M., Venema, Y.: *Modal logic*. Cambridge University Press (2011)
- [8] Blom, S., Van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) *Computer Aided Verification*. LNCS, vol. 6174, pp. 354–359. Springer (2010). doi:10.1007/978-3-642-14295-6_31
- [9] Botinčan, M., Parkinson, M., Schulte, W.: Separation logic verification of C programs with an SMT solver. *Electronic Notes in Theoretical Computer Science* **254**, 5–23 (2009). doi:10.1016/j.entcs.2009.09.057
- [10] Bradfield, J., Walukiewicz, I.: The mu-calculus and model checking. In: *Handbook of Model Checking*, pp. 871–919. Springer (2018). doi:10.1007/978-3-319-10575-8_26
- [11] Broy, M.: Toward a mathematical foundation of software engineering methods. *IEEE Transactions on Software Engineering* **27**(1), 42–57 (2001). doi:10.1109/32.895987

- [12] Bruce, K.B.: Foundations of object-oriented languages: types and semantics. MIT Press (2002)
- [13] Chen, Y.: Checking internal consistency of SOFL specification: A hybrid approach. In: Liu, S., Duan, Z. (eds.) Structured Object-Oriented Formal Language and Method. LNCS, vol. 8332, pp. 175–191. Springer (2014). doi:10.1007/978-3-319-04915-1_13
- [14] Chen, Y., Liu, S., Wang, L.: An extension to data-flow-oriented formal specification language for specifying concurrent software systems. In: QSIC '10 Proceedings of the 2010 10th International Conference on Quality Software. pp. 214–219. IEEE (2010). doi:10.1109/qsic.2010.49
- [15] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: LNCS, pp. 23–42. Springer (2009). doi:10.1007/978-3-642-03359-9_2
- [16] Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., Vink, E.P.D., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 7795, pp. 199–213. Springer (2013). doi:10.1007/978-3-642-36742-7_15
- [17] Dennis, J.B.: Data flow supercomputers. *Computer* **13**(11), 48–56 (1980)
- [18] Dijkstra, E.W.: Guarded commands, nondeterminacy, and formal derivation of programs, pp. 166–175. Springer (1978). doi:10.1007/978-1-4612-6315-9_14
- [19] Dong, J.S., Liu, S.: An object semantic model of SOFL. In: Araki, K., Galloway, A., Taguchi, K. (eds.) IFM'99. pp. 189–208. Springer (1999). doi:10.1007/978-1-4471-0851-1_11
- [20] Dong, J., Liu, S.: The semantics of extended SOFL. In: Proceedings 26th Annual International Computer Software and Applications. pp. 653–658. IEEE (2002). doi:10.1109/CMPSAC.2002.1045077
- [21] Duke, R., Rose, G., Smith, G.P.: Object-Z: A specification language advocated for the description of standards. *Computer Standards & Interfaces* **17**(5-6), 511–533 (Sep 1995)
- [22] Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV'2014). LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
- [23] Fitzgerald, J.S., Jones, C.B.: The connection between two ways of reasoning about partial functions. *Information Processing Letters* **107**(3-4), 128–132 (2008). doi:10.1016/j.ipl.2008.02.005
- [24] Gabow, H., Maheshwari, S., Osterweil, L.: On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering* **2**(3), 227–231 (1976). doi:10.1109/tse.1976.233819

- [25] Gao, X., Miao, H., Chen, Y.: SOZL language: A new software development methodology. In: Proceedings of Conference on Software: Theory and Practice, 16th IFIP World Computer Congress. (2000)
- [26] Gao, X., Miao, H., Liu, S., Liu, L.: The availability semantics of predicate data flow diagram. In: Li, M., Sun, X.H., Deng, Q., Ni, J. (eds.) Grid and Cooperative Computing. LNCS, vol. 3033, pp. 970–977. Springer (2004). doi:10.1007/978-3-540-24680-0_152
- [27] Gibbins, P.F.: VDM: Axiomatising its propositional logic. *Computer Journal* **31**(6), 510–516 (1988). doi:10.1093/comjnl/31.6.510
- [28] Groote, J.F., Mousavi, M.R., Reniers, M.: Modelling and analysis of communicating systems. Technische Universiteit Eindhoven (2013)
- [29] Haugen, Ø.: MSC-2000 interaction diagrams for the new millennium. *Computer Networks* **35**(6), 721–732 (2001). doi:10.1016/s1389-1286(00)00201-2
- [30] Ho-Stuart, C., Liu, S.: A formal operational semantics for SOFL. In: APSEC '97 Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference. pp. 52–61. IEEE (1997). doi:10.1109/APSEC.1997.640161
- [31] Jensen, K.: Coloured Petri Nets. In: Petri Nets: central models and their properties. LNCS, vol. 254, pp. 248–299. Springer (1987). doi:10.1007/978-3-540-47919-2_10
- [32] Jones, C.B.: A small language definition. *Case Studies in Systematic Software Development* pp. 235–256 (1990)
- [33] Jones, C.B., Middelburg, C.A.: A typed logic of partial functions reconstructed classically. *Acta Informatica* **31**(5), 399–430 (1994). doi:10.1007/bf01178666
- [34] Jones, C.B.: Reasoning about partial functions in the formal development of programs. *Electronic Notes in Theoretical Computer Science* **145**, 3–25 (2006). doi:10.1016/j.entcs.2005.10.002
- [35] Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer (2012). doi:10.1007/978-3-642-27919-5
- [36] Li, C., Li, M., Liu, S., Nakajima, S.: Applying “Functional Scenario-Based” test case generation method in unit testing and integration testing. In: Structured Object-Oriented Formal Language and Method. vol. 6, pp. 1–11. Springer (2013). doi:10.1007/978-3-642-39277-1_1
- [37] Li, C., Liu, S., Nakajima, S.: An experiment for assessment of a “Functional Scenario-based” test case generation method. *International Journal of Computer, Electrical, Automation, Control and Information Engineering* **6**(4), 424–431 (2012). doi:10.1999/1307-6892/7240

- [38] Li, M., Liu, S.: Automated functional scenarios-based formal specification animation. In: 2012 19th Asia-Pacific Software Engineering Conference. vol. 1, pp. 107–115. IEEE (2012). doi:10.1109/APSEC.2012.115
- [39] Li, M., Liu, S.: Tool support for rigorous formal specification inspection. In: 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE). pp. 729–734. IEEE (2014). doi:10.1109/cse.2014.151
- [40] Liu, S.: Internal consistency of FRSM specifications. *Journal of Systems and Software* **29**(2), 167–175 (1995). doi:10.1016/0164-1212(94)00073-v
- [41] Liu, S.: A formal definition of FRSM and applications. *International Journal of Software Engineering and Knowledge Engineering* **8**(2), 253–281 (1998). doi:10.1142/s0218194098000145
- [42] Liu, S.: A simulation approach to verification and validation of formal specifications. In: First International Symposium on Cyber Worlds, 2002. Proceedings. pp. 113–120. IEEE (2002). doi:10.1109/cw.2002.1180867
- [43] Liu, S.: A Case Study of Modeling an ATM Using SOFL. Tech. rep., University Hosei (2003)
- [44] Liu, S.: A rigorous approach to reviewing formal specifications. In: 27th Annual NASA Goddard / IEEE Software Engineering Workshop, SEW 2002. pp. 75–81. IEEE (2003). doi:10.1109/SEW.2002.1199452
- [45] Liu, S.: Utilizing specification testing in review task trees for rigorous review of formal specifications. In: Tenth Asia-Pacific Software Engineering Conference, 2003. pp. 510–519. IEEE (2003). doi:10.1109/APSEC.2003.1254406
- [46] Liu, S.: Formal engineering for industrial software development, vol. 3308. Springer (2004). doi:10.1007/978-3-662-07287-5
- [47] Liu, S.: Integrating top-down and scenario-based methods for constructing software specifications. In: 2008 The Eighth International Conference on Quality Software. vol. 5, pp. 105–113. IEEE (2008). doi:10.1109/QSIC.2008.39
- [48] Liu, S.: Extending operation semantics to enhance the applicability of formal refinement. In: Iida, S., Meseguer, J., Ogata, K. (eds.) *Specification, Algebra, and Software*. vol. 8373, pp. 434–440. Springer (2014). doi:10.1007/978-3-642-54624-2_21
- [49] Liu, S., Asuka, M., Komaya, K., Nakamura, Y.: Applying SOFL to specify a railway crossing controller for industry. In: Proceedings 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques. pp. 16–27. IEEE (1998). doi:10.1109/wift.1998.766294
- [50] Liu, S., Chen, Y., Nagoya, F., McDermid, J.A.: Formal specification-based inspection for verification of programs. *IEEE Transactions on Software Engineering* **38**(5), 1100–1122 (2012). doi:10.1109/TSE.2011.102

- [51] Liu, S., Dong, J.S., Jin Song Dong, J.S.: Extending SOFL to support both top-down and bottom-up approaches. In: IEEE International Conference on Systems, Man and Cybernetics. vol. 6, pp. 6–pp. IEEE (2002). doi:10.1109/ICSMC.2002.1175573
- [52] Liu, S., Jin Song Dong, J., Dong, J.S.: Class and module in SOFL. In: Proceedings Second Asia-Pacific Conference on Quality Software. pp. 241–245. IEEE (2001). doi:10.1109/APAQS.2001.990026
- [53] Liu, S., McDermid, J.A., Chen, Y.: A rigorous method for inspection of model-based formal specifications. IEEE Transactions on Reliability **59**(4), 667–684 (2010). doi:10.1109/TR.2010.2085571
- [54] Liu, S., Nagoya, F., Chen, Y., Goya, M., McDermid, J.A.: An automated approach to specification-based program inspection. In: Lau, K.K., Banach, R. (eds.) Formal Methods and Software Engineering. LNCS, vol. 3785, pp. 421–434. Springer (2005). doi:10.1007/11576280_29
- [55] Liu, S., Nakajima, S.: A decompositional approach to automatic test case generation based on formal specifications. In: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. pp. 147–155. IEEE (2010). doi:10.1109/ssiri.2010.11
- [56] Liu, S., Shibata, M., Sato, R.: Applying SOFL to develop a university information system. In: ASPEC'99 Proceedings Sixth Asia-Pacific Software Engineering Conference. pp. 404–411. IEEE (1999). doi:10.1109/apsec.1999.809630
- [57] Liu, S., Sun, Y.: Structured methodology+object-oriented methodology+formal methods: methodology of SOFL. In: Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'95. pp. 137–144. IEEE (1995). doi:10.1109/iceccs.1995.479319
- [58] Liu, S., Wang, H.: An automated approach to specification animation for validation. Journal of Systems and Software **80**(8), 1271–1285 (2007). doi:10.1016/j.jss.2006.12.540
- [59] Liu, S., Woodcock, J.: Supporting rigorous reviews of formal specifications using fault trees. In: Proceedings of Conference on Software: Theory and Practice, 16th World Computer Congress 2000. pp. 459–470 (2000)
- [60] Milner, R.: A complete inference system for a class of regular behaviours. Journal of Computer and System Sciences **28**(3), 439–466 (1984). doi:10.1016/0022-0000(84)90023-0
- [61] Moura, L.D., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008). doi:10.1007/978-3-540-78800-3_24
- [62] Offutt, A., Xiong, Y., Liu, S.: Criteria for generating specification-based tests. In: ICECCS'99 Proceedings Fifth IEEE International Conference on Engineering of Complex Computer Systems. pp. 119–129. IEEE (1999). doi:10.1109/iceccs.1999.802856

- [63] Offutt, A., Liu, S.: Generating test data from SOFL specifications. *Journal of Systems and Software* **49**(1), 49–62 (1999). doi:10.1016/s0164-1212(99)00066-7
- [64] Paar, A., Gruner, S.: Static typing with value space-based subtyping. In: SAICSIT '11 Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment. ACM Press (2011). doi:10.1145/2072221.2072242
- [65] Parr, T.: *The Definitive ANTLR 4 Reference*. O'Reilly UK Ltd. (2015)
- [66] Plotkin, G.D.: The origins of structural operational semantics. *Journal of Logic and Algebraic Programming* **60-61**, 3–15 (2004). doi:10.1016/j.jlap.2004.03.009
- [67] Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. *Journal of Functional Programming* **20**(1), 71 (2010). doi:10.1017/s0956796809990293
- [68] Stappers, F.P.M., Reniers, M.A., Groote, J.F.: Suitability of mCRL2 for concurrent-system design: A 2×2 switch case study. In: *Formal Methods for Components and Objects*. pp. 166–185. Springer (2010). doi:10.1007/978-3-642-17071-3_9
- [69] Steyn, T.J., Gruner, S.: A new optional parallelism operator in CSP for wireless sensor networks. In: SAICSIT '17 Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment. ACM Press (2017). doi:10.1145/3129416.3129431
- [70] Tian, C., Liu, S., Duan, Z.: Abstract model checking with SOFL hierarchy. In: *Structured Object-Oriented Formal Language and Method, SOFL 2012* (2012). doi:10.1007/978-3-642-39277-1_6
- [71] Van Eekelen, M., Ten Hoedt, S., Schreurs, R., Usenko, Y.S.: Analysis of a session-layer protocol in mCRL2. In: *Formal Methods for Industrial Critical Systems*, pp. 182–199. Springer Berlin Heidelberg (2008). doi:10.1007/978-3-540-79707-4_14
- [72] Wadler, P.: Propositions as types. *Communications of the ACM* **58**(12), 75–84 (Nov 2015). doi:10.1145/2699407
- [73] Wang, J., Liu, S., Qi, Y., Hou, D.: Developing an insulin pump system using the SOFL method. In: *14th Asia-Pacific Software Engineering Conference (APSEC'07)*. pp. 334–341. IEEE (2007). doi:10.1109/aspec.2007.31
- [74] Wang, X., Liu, S.: Development of a supporting tool for formalizing software requirements. In: *Structured Object-Oriented Formal Language and Method, SOFL 2012*. pp. 56–70 (2012). doi:10.1007/978-3-642-39277-1_5
- [75] Xiaolei, G., Miao, H.: The axiomatic semantics of PDFD. In: *2008 Japan-China Joint Workshop on Frontier of Computer Science and Technology*. pp. 139–146. IEEE (2008). doi:10.1109/fcst.2008.18

Appendix A

Grammar

This chapter defines the grammar of the SOFL dialect used in this study using EBNF enriched with regular expressions and using a custom operation `<name>`. The operation `<name>` is a semantics element but is included in the syntax since referencing of previously defined element is of such importance; this also gives a “flavour” that each rule has a specific type. Two string elements are used:

1. ID: a string used to name an element and
2. String: string for general use.

To convert the grammar to a form that does not use the semantic information of referencing, `<name>` is replaced by a string rule that starts on a letter and does not contain spaces.

Differences between the grammar of the dialect and SOFL are indicated by colour: **Green** is used to show that an EBNF rule changed considerably and **Red** is used to show that there is a change to part of a rule.

A.1 Module and CDFD Syntax

```
SoflModule := "module" ID ["/" <SoflModule>]
  [ConstantDecl] [TypeDecl]
  [VariablesDecl] [InvariantDecl]
  [SoflBehaviour] [SoflInitProcess]
  SoflProcess*
  SoflFunction*
  "end_module"

ConstantDecl := "const" Constant (";" Constant)*

Constant := SoflVariableName "=" Expression

TypeDecl := "type" NewType (";" NewType)*

VariablesDecl := "var" (NewVariable |"ext" ExternalVariable)
  ( ";"(NewVariable|"ext" ExternalVariable))+
```

```
InvariantDecl := "inv" Expression (";" Expression)*
```

```
SoflInitProcess := "process" "Init" "(" ")"
  Expression (";"Expression)*
  [Comment]
  "end_process"
```

A summary of the changes:

Constant: The **Expression** can only return a value and is by default a constant value since the expression is evaluated on initialisation of the module.

InvariantDecl: The type of **Expression** need to verified to be boolean.

SoflInitProcess: Only syntax specialised to the **Init** process is defined and the **Expression** define the initialisation state of the module.

VariablesDecl: The structure of the syntax definition is different to assist in the Xtext. The keyword **rd** and **wr**, no variable outside the system is allowed.

A.2 Behaviour

```
SoflBehaviour := "behaviour" ID "(" DataFlows ")" DataFlows
  [SoflAdding SoflGrouping]
  StateConditionsDefinition (
    ConditionStructure | BroadcastStructure |
    MergeStructure | UnMergeStructure
  )*
  (Flow | ShaddowFlow)*
  "end_behaviour"
```

```
SoflGrouping := "grouping" "(" DataFlowReference ")"
  DataFlowReference
```

```
SoflAdding := "added" "(" (PortDeclarations|"void") ")"
  (PortDeclarations|"void")
```

```
StateConditionsDefinition := Statecondition+
```

```
ConditionStructure := "condition" ID (
  "(" SoflVariableName ")"
  CondOption ("," CondOption)*
  )"end_condition"
```

```
CondOption := SoflVariableName ":" Expression
```

```
BroadcastStructure := "broadcast" ID "("
  ((
    SoflVariableName ")" SoflVariableName ("," SoflVariableName)*
  )|(
```

```

    ConnectionName ")"
    ConnectionName ("," ConnectionName)*
  )"end_broadcast"

MergeStructure := "merge" ID (
  "(" SoflVariableName ("," SoflVariableName)* ")"
  SoflVariableName
)"end_merge"

UnMergeStructure := "un_merge" ID(
  "(" SoflVariableName ")"
  SoflVariableName ("," SoflVariableName)*
)"end_unmerge"

Flow := "flow" FlowSource "=>" FlowDestination

FlowSource := NodeName "." <SoflVariableName>

FlowDestination := NodeName "." <SoflVariableName>

ShaddowFlow := "shadow_flow" ShaddowFlowSourceNode
              "=>" ShaddowFlowDestinationNode

ShaddowFlowSourceNode := NodeName "." <ConnectionName>

ShaddowFlowDestinationNode := NodeName "." <ConnectionName>

NodeName := "behavior" |
  "cond" <ConditionStructure> |
  "proc" <SoflProcess> |
  "cast" <BroadcastStructure> |
  "merge" <MergeStructure> |
  "unmerge" <UnMergeStructure>

```

The definition of **SoflBehaviour** is new and all the element that it references. This is added to replace a graphical representation of a CDFD.

A.3 Process and Port Syntax

```

SoflProcess := "process" ID "(" DataFlows ")" DataFlows
  [ProcessExternalVariable]
  StateConditionsDefinition
  [Decomposition] [Comment]
"end_process"

```

```

PortDeclarations := (VariableDeclare|ConnectionName)
  (","(
  VariableDeclare|

```


ConnectionName))*

ProcessExternalVariable := "ext" ProcessVariable+

ProcessVariable := ("rd" | "wr") <SoflVariableName>

DataFlows := PortDeclarations ("|" PortDeclarations)*

DataFlowReference := PortReference ("|" PortReference)*

```
PortReference := (
  <SoflVariableName> |
  "connection" <ConnectionName>
)(", "
  (<SoflVariableName>|
  "connection" <ConnectionName>)
)*
```

Comment := "comment" STRING

A summary of the changes:

PortReference: The syntax for a shadow flow is added.

SoflProcess: An explicit specification is not allowed. **StateConditionsDefinition** is new and is used to specify the pre and post conditions of the process.

A.4 Type Syntax

NewType := TypeIdentifier "=" SoflType

SoflType := TypeLower ("*" TypeLower)*

```
TypeLower :=<TypeIdentifier> |
  "nat0"|"nat"|"int"|"real"|"char"|"bool"|"string"|
  Enumeration|ComposedType|SequenceType|SetType|MapType
```

MapType := "map" SoflType "to" SoflType

SetType := "set" "of" SoflType

SequenceType := "seq" "of" SoflType

```
ComposedType := "composed" "of" [<TypeIdentifier> "and"]
  ComposedElement+
  "end"
```

ComposedElement := ID ":" SoflType

```
Enumeration := "{" EnumerateValue ("," EnumerateValue)* "}"
```

```
EnumerateValue := "<" ID ">"
```

Types are removed from `TypeLower` as specified in Section 3.1.5.

A.5 Expression and Function Syntax

```
BinaryOperation := "or"|"and"|"=">"|"<=">"|"="|"<>"|
  "notin"|"<"|">"|"<="|">="|"+"|"-|"*"|"**"|
  "/"|"div"|"rem"|"mod"
```

```
Expression :=
```

```
(Expression BinaryOperation Expression)*|
("let" PatternDefinition
  ("," PatternDefinition)* "in" Expression)|
("if" Expression "then" Expression "else" Expression)|
(
  (
    ("forall" | "exists" ["!"])
      [" PortDeclarations "|" Expression "]") |
    (
      [" (Expression(
        [(", " Expression)*)|
        ("...", " Expression)|
        ("|" NewVariable
          ("," NewVariable) "&" Expression)
      )
      ]]" ["as" SoflType]
    )|
    ("{"(
      MapSequence|
      CompressionExpression|
      ExpressionSequence|
      "->")
      }")|
    ([ "~" ]<SoflVariableName> ["("ExpressionSequence")"])|
    PreDefinedFunctions|
    ("("Expression ")")|
    ("case" CaseExpression "end_case")
  )FunctionReferenceModifiers*)|
"<"[EnumerateValue]">"|
STRING | ("-" Expression)
| "true" | "false" | ("not" Expression)
| SoflNumber
```

```
PreDefinedFunctions := (
  (
    "card"|"dom"|"rng"|"subset"|
    "psubset"|"len"|"union"|
```

```

    "inter"|"dinter"|"dunion"|
    "mk_"[TypeIdentifier]|
    "override"|"domrb"|"diff"|
    "get"|"comp"|"power"|"conc"|
    "dconc"|"hd"|"tl"|"domrt"|
    "rngrt"|"rngrb"|"domdl"|
    "floor"|"bound"|"elems"|"inds"|
    "inverse"|"is_"<TypeIdentifier>|"abs"
  )("ExpressionSequence ")
)|
"modify" ("Expression", ArgumentsModifyItem
  (" ArgumentsModifyItem)* ")
ArgumentsModifyItem := (<ComposedElement>|Expression)
  "->" Expression

NewVariable := SoflVariableName ":" SoflType

ExternalVariable := ["#"] ("rd" | "wr") (
  <SoflVariableName> |
  SoflVariableName ":" SoflType
)

FunctionReferenceModifiers := (" ExpressionSequence ") |
  "."<ComposedElement>

VariableDeclare := SoflVariableName (" SoflVariableName)*
  ":" SoflType

VariableDeclareList := VariableDeclare
  (" VariableDeclare)*

ConnectionName := "connect" ID

PatternDefinition := SoflVariableName(
  ("=" Expression)|
  (":" (SoflType "|" Expression))
)

CaseExpression := Expression "of" CaseAlternative
  ( ";" CaseAlternative)*
  [ ";" "default" "->" Expression]

CaseAlternative := Expression
  (" Expression)* "->" Expression

ExpressionSequence := Expression (" Expression)*

CompressionExpression := Expression "|"

```

```

    (VariableDeclareList "&" Expression)

MapSequence := MapElement ("," MapElement)*

MapElement := Expression "->" Expression

ParamSingleTypeDecl := SoflVariableName
    ("," SoflVariableName)*
    ":" SoflType

ParamDecl := ParamSingleTypeDecl ("," ParamSingleTypeDecl)*

SoflVariableName := ID

SoflFunction := "function" SoflVariableName "(" ParamDecl ")"
    SoflVariableName ":" SoflType
    PreCondition PostCondition ["===" Expression]
    "end_function"

PreCondition := "pre" Expression

PostCondition := "post" Expression

Statecondition := "state_condition" "in" SoflNumber
    "out" SoflNumber PreCondition PostCondition

Decomposition := "decomposition" <SoflModule>

TypeIDentifier := ID

SoflNumber := ["-"] Digits ["." Digits]
    ["nat0"|"nat"|"int"|"real"]

Digits := "0"|"1"| ...|"9"

```

Decomposition Specifies the CDFD that refines a process by making reference to the name of the CDFD.

Expression: Expression are modified so that they cannot change state and the syntax is therefore changed to accommodate the required semantic change.

SoflFunction An explicit name of the variable returned is given by **SoflVariableName** and it is compulsory to specify the pre and post condition.

SoflNumber An optional type annotation is added.

Statecondition This is added to specify the pre and post condition for processes and behaviours.

Appendix B

Example Specification

This is a specification based on the example in Section 6.2.4.

B.1 Small Example

B.1.1 Specifications

```

module SYSTEM_Parent_small
  var parent_var:int
  inv parent_var > 10 and parent_var < 40

  behaviour Parent_CDFD(in_1:int | connect alt) connect oo | out_bottom:nat0
    state_condition in 1 out 1 pre in_1>10 post true
    state_condition in 1 out 2 pre in_1>=10 post true
    state_condition in 2 out 1 pre false post true
    state_condition in 2 out 2 pre false post true
    flow behaviour.in_1 => proc ToRefine.in_1
    flow proc ToRefine.out_bottom => behaviour.out_bottom
    shadow_flow proc ToRefine.oo => behaviour.oo
    shadow_flow behaviour.alt => proc ToRefine.alt
  end_behaviour
  process Init()
    parent_var < 100
    and parent_var > 0
  end_process
  process ToRefine(in_1:int | connect alt) connect oo | out_bottom:nat0
    ext wr parent_var
    state_condition in 1 out 1
      pre (in_1 > 10) and(in_1 < parent_var) post true
    state_condition in 1 out 2
      pre in_1 > 0 post parent_var < out_bottom + 50
    state_condition in 2 out 1
      pre parent_var > 10 and parent_var< 50 post parent_var > 40
    state_condition in 2 out 2
      pre true post (parent_var < out_bottom + 50) and (parent_var > 10)
  end_process
end_module

```

```

    decomposition SYSTEM_Child_small
  end_process
end_module

```

Listing B.1: The top module of the small example

```

module SYSTEM_Child_small/SYSTEM_Parent_small
  var ss1:int;
      ss2:int;
      ss3:int;
      ss4:int

  inv ss1 > 9;
      ss1 < ss2;
      ss2 < ss3;
      ss3 < 200

  behaviour Child_CDFD(in_1: int | connect alt) connect oo | out_bottom:nat0
    added(in_v:int) out_top:int,other_bottom:nat0
    grouping(in_1 | connection alt , in_v)
      connection oo, out_top | out_bottom, other_bottom

    state_condition in 1 out 1 pre in_1 > 0 post out_top > 0
    state_condition in 1 out 2 pre in_1 > 0 post out_bottom > 0
    state_condition in 2 out 1 pre true post out_top > 0
    state_condition in 2 out 2 pre true post out_bottom > 10

    condition ConditionNode(in_1)
      out_loop: out_loop > 10,
      out_noloop: out_noloop <= 10
    end_condition

  broadcast BroadCastNode(in_v) c_data_1, c_data_2, c_data_3 end_broadcast

  shadow_flow behaviour.alt => proc P_first.alt
  flow behaviour.in_1 => cond ConditionNode.in_1
  flow behaviour.in_v => cast BroadCastNode.in_v
  flow cast BroadCastNode.c_data_1 => proc P_noloop.c_data_1
  flow cast BroadCastNode.c_data_2 => proc P_first.c_data_2
  flow cast BroadCastNode.c_data_3 => proc P_last.c_data_3
  flow cond ConditionNode.out_loop => proc P_loop.out_loop
  flow cond ConditionNode.out_noloop => proc P_noloop.out_noloop
  flow proc P_loop.loop_done => proc P_cont.loop_done
  flow proc P_loop.loop_fallout => proc P_cont.loop_fallout
  flow proc P_loop.loop => proc P_loop.loop
  flow proc P_cont.out_top => behaviour.out_top
  shadow_flow proc P_cont.oo => behaviour.oo
  flow proc P_first.other_bottom => behaviour.other_bottom
  flow proc P_first.to_second => proc P_second.to_second
  flow proc P_second.to_last => proc P_last.to_last

```

```

flow proc P_noloop.d_data => proc P_cont.d_data
flow proc P_last.out_bottom => behaviour.out_bottom

end_behaviour

process Init()
  ss3 < 20;
  ss3 > 11
end_process

process P_loop(out_loop:int | loop:int) loop_done:int | loop,loop_fallout:int
  ext wr ss3
  state_condition in 1 out 1 pre out_loop > 10 post loop_done = ~out_loop
  state_condition in 1 out 2 pre out_loop > 5
    post loop = ~out_loop + 1 and loop_fallout > 45
  state_condition in 2 out 1 pre loop > 10 post loop_done = ~loop
  state_condition in 2 out 2 pre loop > 0
    post loop = ~loop + 2 and loop_fallout > 50
end_process

process P_noloop(out_noloop:int | c_data_1:int) d_data:bool

  ext wr ss3
  rd ss2
  state_condition in 1 out 1
    pre out_noloop < 30 post d_data = true and d_data = (~out_noloop < 40)
  state_condition in 2 out 1
    pre c_data_1 > 15 post d_data = false and d_data = (~c_data_1 < 10)
end_process

process P_cont(loop_fallout:int| loop_done:int| d_data:bool)
  connect oo, out_top:int
  ext wr ss3
  state_condition in 1 out 1
    pre loop_fallout > 20 post out_top > 30 and out_top > ~loop_fallout
  state_condition in 2 out 1
    pre loop_done > 20 post out_top > 20
  state_condition in 3 out 1
    pre d_data or not d_data post out_top > 30
end_process

process P_first(connect alt, c_data_2:int) other_bottom:nat0, to_second:int
  ext wr ss3
  state_condition in 1 out 1
    pre ss3 > 11 post other_bottom > 10 and to_second < 20 and to_second > 10
end_process

process P_second(to_second:int) to_last:nat0
  ext wr ss3

```

```

state_condition in 1 out 1
  pre to_second < 20 and to_second > 9 post to_last > (2 * ~to_second)
end_process

process P_last(c_data_3:int, to_last:nat0) out_bottom:nat0
  ext wr ss3
  state_condition in 1 out 1
  pre ss3 < 100 and to_last > 17 post ~to_last > 17
end_process
end_module

```

Listing B.2: The module that refined a process in the small example

B.1.2 mCRL2 Description

```

% The data flow id's used
sort FlowId = struct Child_CDFDin_1_in_1ConditionNode |
  Child_CDFDin_v_in_vBroadCastNode |
  BroadCastNodec_data_1_c_data_1P_noloop |
  BroadCastNodec_data_2_c_data_2P_first |
  BroadCastNodec_data_3_c_data_3P_last |
  ConditionNodeout_loop_out_loopP_loop |
  ConditionNodeout_noloop_out_noloopP_noloop |
  P_looploop_done_loop_doneP_cont |
  P_looploop_fallout_loop_falloutP_cont |
  P_looploop_loopP_loop |
  P_contout_top_out_topChild_CDFD |
  P_firstother_bottom_other_bottomChild_CDFD |
  P_firstto_second_to_secondP_second |
  P_secondto_last_to_lastP_last |
  P_noloopd_data_d_dataP_cont |
  P_lastout_bottom_out_bottomChild_CDFD |
  Child_CDFDalt_altP_first | P_contoo_ooChild_CDFD;
% The node id's of the nodes in the diagram
sort NodeId = struct cdfd_node | NIDChild_CDFD |
  NIBroadCastNode | NIConditionNode | NIP_cont |
  NIP_first | NIP_last | NIP_loop | NIP_noloop | NIP_second;
% Indexes used to identify ports
sort PortId = struct IPO | IP1 | IP2;
% Id's of the data stores used
sort StoreId = struct non_store | ss1 | ss2 | ss3 | ss4;
% Used to identify the type of node
sort NodeType = struct type_control | type_process;
sort FlowIdAction = struct flow_generate | flow_consume |
  flow_query;
% The phase to execute
sort Phase = struct exec_control | exec_process;
sort Rights = struct read_access_begin | read_access_done |
  write_access_begin | write_access_done | rights_done;
sort StoreMap = StoreId -> Int;

```



```

map
flip_phase: Phase ->Phase;
init_store:StoreId->Int;
%%
% Diagram information functions
%%
% The input flow of the digram and a specific input port
cdfdInputFlowIds: PortId -> FSet(FlowId);
% The output flow of the digram and a specific input port
cdfdOutputFlowIds: PortId -> FSet(FlowId);
% the input ports of the diagram
cdfd_inputPort: FSet(PortId);
% the output ports of the diagram
cdfd_outputPort: FSet(PortId);
% get list of ports that are allowed to fire for a specific node
get_list_of_fire_ports:NodeId # Set(FlowId) -> List(PortId);
%%
% Node structure functions
%%
% the inflow of a node
nodeInFlowIds: NodeId -> FSet(FlowId);
% the stores the process access with read access
processStoresRead: NodeId -> FSet(StoreId);
% the stores the process access with write access
processStoresWrite: NodeId -> FSet(StoreId);
io_relation: NodeId # PortId -> FSet(PortId);
% get the nodes that are allowrd to execute in the given phase
getNodeOfType: Phase -> List(NodeId);
% the connected in flows to a node and port
inPortDataFlowId: NodeId # PortId -> FSet(FlowId);
% the connected out flows to a node and port
outPortDataFlowId: NodeId # PortId -> FSet(FlowId);
% the input ports of a node
nodeInPorts: NodeId -> List(PortId);
% the output ports of a node
nodeOutPorts: NodeId -> List(PortId);
%%
% Node fire functions
%%
% determine is node is allowed to fire
can_fire: NodeId # FSet(FlowId) -> Bool;
% get the sublift of nodes that are allowed to fire
get_fire_subset: List(NodeId) # FSet(FlowId) # Phase -> List(NodeId);
% determine if a port can consume data
can_consume: NodeId # FSet(FlowId)-> Bool;
% determine is a port of a node can fire
can_port_consume: NodeId # PortId # FSet(FlowId)-> Bool;
% determine is a port of a node can generate data

```

```

can_port_generate: NodeId # PortId # FSet(FlowId)-> Bool;
% determine if node can generate data
can_generate: NodeId # FSet(FlowId) -> Bool;
% filter ports of node that are allowed to fire
filter_fire_ports: NodeId # Set(FlowId) # List(PortId) # List(PortId)
    -> List(PortId);
%%
% Store functions
%%
store_rights: StoreId -> Rights;
% update the store access by adding the interger parameter
update_access_count: StoreId # StoreMap # Int -> StoreMap;
node_read_list: NodeId -> List(StoreId);
node_write_list: NodeId -> List(StoreId);

var
v_phase: Phase;
n: StoreId;
int_var: Int;
store_map: StoreMap;
flw: FlowId;
node_type: Phase;
n_list1: List(NodeId);
f_list1: List(FlowId);
f_list2: List(FlowId);
f_list3: List(FlowId);
f_set1: FSet(FlowId);
f_set2: FSet(FlowId);
f_set3: FSet(FlowId);
sid: StoreId;
store_set: FSet(StoreId);
store_list: FSet(StoreId);
pid: PortId;
nd_id: NodeId;
prt_list_out: List(PortId);
prt_lst_in: List(PortId);

eqn
(exec_process == v_phase) ->
    flip_phase(v_phase) = exec_control;
(exec_control == v_phase) ->
    flip_phase(v_phase) = exec_process;

get_list_of_fire_ports(nd_id, f_set1) = filter_fire_ports(
    nd_id, f_set1,
    nodeInPorts(nd_id), []);

(#prt_lst_in == 0) ->
    filter_fire_ports(nd_id,

```

```

    f_set1,
    prt_lst_in,
    prt_list_out) = prt_list_out;

(#prt_lst_in > 0 &&
    can_port_consume(nd_id, head(prt_lst_in), f_set1)) ->
    filter_fire_ports(nd_id, f_set1, prt_lst_in, prt_list_out) =
    filter_fire_ports(nd_id,
        f_set1, tail(prt_lst_in),
        [head(prt_lst_in)] ++ prt_list_out);

(#prt_lst_in > 0 &&
    !can_port_consume(nd_id, head(prt_lst_in), f_set1)) ->
    filter_fire_ports(nd_id, f_set1, prt_lst_in, prt_list_out) =
    filter_fire_ports(nd_id, f_set1, tail(prt_lst_in), prt_list_out);

(true) -> init_store(n)=0;
(NIBroadCastNode == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        {Child_CDFDin_v_in_vBroadCastNode};
(NIConditionNode == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        {Child_CDFDin_1_in_1ConditionNode};
(NIP_cont == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        {P_looploop_fallout_loop_falloutP_cont};
(NIP_cont == nd_id && IP1 == pid) ->
    inPortDataFlowId(nd_id, pid) =
        {P_looploop_done_loop_doneP_cont};
(NIP_cont == nd_id && IP2 == pid) ->
    inPortDataFlowId(nd_id, pid) = {P_noloopd_data_d_dataP_cont};
(NIP_first == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        { BroadcastNodec_data_2_c_data_2P_first,
        Child_CDFDalt_altP_first};
(NIP_last == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        { BroadcastNodec_data_3_c_data_3P_last,
        P_secondto_last_to_lastP_last};
(NIP_loop == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        {ConditionNodeout_loop_out_loopP_loop};
(NIP_loop == nd_id && IP1 == pid) ->
    inPortDataFlowId(nd_id, pid) = {P_looploop_loopP_loop};
(NIP_noloop == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        { ConditionNodeout_noloop_out_noloopP_noloop};
(NIP_noloop == nd_id && IP1 == pid) ->
    inPortDataFlowId(nd_id, pid) =

```

```

        {BroadCastNodec_data_1_c_data_1P_noloop};
(NIP_second == nd_id && IPO == pid) ->
    inPortDataFlowId(nd_id, pid) =
        {P_firstto_second_to_secondP_second};

(NIBroadCastNode == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {BroadCastNodec_data_1_c_data_1P_noloop,
         BroadCastNodec_data_2_c_data_2P_first,
         BroadCastNodec_data_3_c_data_3P_last};
(NIConditionNode == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {ConditionNodeout_loop_out_loopP_loop};
(NIConditionNode == nd_id && IP1 == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {ConditionNodeout_noloop_out_noloopP_noloop};
(NIP_cont == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {P_contoo_ooChild_CDFD,P_contout_top_out_topChild_CDFD};
(NIP_first == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {P_firsttother_bottom_other_bottomChild_CDFD,
         P_firstto_second_to_secondP_second};
(NIP_last == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {P_lastout_bottom_out_bottomChild_CDFD};
(NIP_loop == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {P_looploop_done_loop_doneP_cont};
(NIP_loop == nd_id && IP1 == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {P_looploop_fallout_loop_falloutP_cont,
         P_looploop_loopP_loop};
(NIP_noloop == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {P_noloopd_data_d_dataP_cont};
(NIP_second == nd_id && IPO == pid) ->
    outPortDataFlowId(nd_id, pid) =
        {P_secondto_last_to_lastP_last};

(NIBroadCastNode == nd_id) -> nodeInFlowIds(nd_id) =
    { Child_CDFDin_v_in_vBroadCastNode };
(NIConditionNode == nd_id) -> nodeInFlowIds(nd_id) =
    { Child_CDFDin_1_in_1ConditionNode };
(NIP_cont == nd_id) -> nodeInFlowIds(nd_id) =
    { P_looploop_done_loop_doneP_cont,
      P_looploop_fallout_loop_falloutP_cont,
      P_noloopd_data_d_dataP_cont };
(NIP_first == nd_id) -> nodeInFlowIds(nd_id) =

```

```

{ BroadCastNodec_data_2_c_data_2P_first,
  Child_CDFDalt_altP_first };
(NIP_last == nd_id) -> nodeInFlowIds(nd_id) =
{ BroadCastNodec_data_3_c_data_3P_last,
  P_secondto_last_to_lastP_last };
(NIP_loop == nd_id) -> nodeInFlowIds(nd_id) =
{ ConditionNodeout_loop_out_loopP_loop,
  P_looploop_loopP_loop };
(NIP_noloop == nd_id) -> nodeInFlowIds(nd_id) =
{ BroadCastNodec_data_1_c_data_1P_noloop,
  ConditionNodeout_noloop_out_noloopP_noloop };
(NIP_second == nd_id) -> nodeInFlowIds(nd_id) =
{ P_firstto_second_to_secondP_second };

(NIBroadCastNode == nd_id) -> node_write_list(nd_id) = [];
(NIConditionNode == nd_id) -> node_write_list(nd_id) = [];
(NIP_cont == nd_id) -> node_write_list(nd_id) = [ ss3 ];
(NIP_first == nd_id) -> node_write_list(nd_id) = [ ss3 ];
(NIP_last == nd_id) -> node_write_list(nd_id) = [ ss3 ];
(NIP_loop == nd_id) -> node_write_list(nd_id) = [ ss3 ];
(NIP_noloop == nd_id) -> node_write_list(nd_id) = [ ss3 ];
(NIP_second == nd_id) -> node_write_list(nd_id) = [ ss3 ];
(NIBroadCastNode == nd_id) -> node_read_list(nd_id) = [];
(NIConditionNode == nd_id) -> node_read_list(nd_id) = [];
(NIP_cont == nd_id) -> node_read_list(nd_id) = [];
(NIP_first == nd_id) -> node_read_list(nd_id) = [];
(NIP_last == nd_id) -> node_read_list(nd_id) = [];
(NIP_loop == nd_id) -> node_read_list(nd_id) = [];
(NIP_noloop == nd_id) -> node_read_list(nd_id) = [ ss2 ];
(NIP_second == nd_id) -> node_read_list(nd_id) = [];

(NIBroadCastNode == nd_id) -> nodeInPorts(nd_id) = [IP0];
(NIConditionNode == nd_id) -> nodeInPorts(nd_id) = [IP0];
(NIP_cont == nd_id) -> nodeInPorts(nd_id) = [IP0,IP1,IP2];
(NIP_first == nd_id) -> nodeInPorts(nd_id) = [IP0];
(NIP_last == nd_id) -> nodeInPorts(nd_id) = [IP0];
(NIP_loop == nd_id) -> nodeInPorts(nd_id) = [IP0,IP1];
(NIP_noloop == nd_id) -> nodeInPorts(nd_id) = [IP0,IP1];
(NIP_second == nd_id) -> nodeInPorts(nd_id) = [IP0];

(NIBroadCastNode == nd_id) -> nodeOutPorts(nd_id) = [IP0];
(NIConditionNode == nd_id) -> nodeOutPorts(nd_id) = [IP0,IP1];
(NIP_cont == nd_id) -> nodeOutPorts(nd_id) = [IP0];
(NIP_first == nd_id) -> nodeOutPorts(nd_id) = [IP0];
(NIP_last == nd_id) -> nodeOutPorts(nd_id) = [IP0];
(NIP_loop == nd_id) -> nodeOutPorts(nd_id) = [IP0,IP1];
(NIP_noloop == nd_id) -> nodeOutPorts(nd_id) = [IP0];

```

```

(NIP_second == nd_id) -> nodeOutPorts(nd_id) = [IP0];
(NIBroadCastNode == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIConditionNode == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0,IP1};
(NIP_cont == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIP_cont == nd_id && IP1 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIP_cont == nd_id && IP2 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIP_first == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIP_last == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIP_loop == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0,IP1};
(NIP_loop == nd_id && IP1 == pid) ->
  io_relation(nd_id, pid) = {IP0,IP1};
(NIP_noloop == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIP_noloop == nd_id && IP1 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIP_second == nd_id && IP0 == pid) ->
  io_relation(nd_id, pid) = {IP0};
(NIChild_CDFD == nd_id) -> nodeInPorts(nd_id) = [IP0,IP1];
(NIChild_CDFD == nd_id) -> nodeOutPorts(nd_id) = [IP0,IP1];
(true) -> cdfd_inputPort = {IP0,IP1};
(true) -> cdfd_outputPort = {IP0,IP1};
(IP0 == pid) -> cdfdInputFlowIds(pid) =
  {Child_CDFDin_1_in_1ConditionNode};
(IP1 == pid) -> cdfdInputFlowIds(pid) =
  {Child_CDFDalt_altP_first,Child_CDFDin_v_in_vBroadCastNode};

(IP0 == pid) -> cdfdOutputFlowIds(pid) =
  {P_contoo_ooChild_CDFD,P_contout_top_out_topChild_CDFD};
(IP1 == pid) -> cdfdOutputFlowIds(pid) =
  {P_firstother_bottom_other_bottomChild_CDFD,
   P_lastout_bottom_out_bottomChild_CDFD};
(node_type == exec_control) -> getNodeOfType(node_type) =
  [NIBroadCastNode,NIConditionNode];
(node_type == exec_process) -> getNodeOfType(node_type) =
  [NIP_cont,NIP_first,NIP_last,NIP_loop,NIP_noloop,NIP_second];
(true) -> processStoresRead(nd_id) = {};
(true) -> processStoresWrite(nd_id) = {};

(nd_id == NIChild_CDFD) -> can_port_consume(nd_id, pid, f_set1) =
  false;
(nd_id == NIChild_CDFD) -> can_port_generate(nd_id, pid, f_set1) =

```

```

false;

(true) -> can_port_consume(nd_id, pid, f_set1) =
  inPortDataFlowId(nd_id, pid) * f_set1 ==
  inPortDataFlowId(nd_id, pid);
(true) -> can_port_generate(nd_id, pid, f_set1) =
  (outPortDataFlowId(nd_id, pid) - nodeInFlowIds(nd_id)) *
  f_set1 == {};

(nd_id == NICchild_CDFD) -> can_fire(nd_id, f_set1) = false;
(true) -> can_fire(nd_id, f_set1) =
  can_consume(nd_id, f_set1) && (can_generate(nd_id, f_set1));

(true) -> can_generate(nd_id, f_set1) =
  forall pid: PortId . pid in nodeOutPorts(nd_id) =>
    can_port_generate(nd_id, pid, f_set1);
(true) -> can_consume(nd_id, f_set1) =
  exists pid: PortId . (pid in nodeInPorts(nd_id)) &&
  can_port_consume(nd_id, pid, f_set1);

(true) -> update_access_count(sid, store_map, int_var) =
  store_map[sid -> store_map(sid) + int_var];

(#n_list1 > 0 && can_fire(head(n_list1), f_set2)) ->
  get_fire_subset(n_list1, f_set2, v_phase) =
  get_fire_subset(tail(n_list1),
    f_set2, v_phase) ++ [head(n_list1)];
(#n_list1 > 0 && !can_fire(head(n_list1), f_set2)) ->
  get_fire_subset(n_list1, f_set2, v_phase) =
  get_fire_subset(tail(n_list1), f_set2, v_phase);
(#n_list1 == 0 ) -> get_fire_subset(n_list1, f_set2, v_phase) = [];

act
error_no_output; % indicate that node output was generated,
  % an error state model/specification translation incorrect
store_check; % synchronised message that indicate store
  % access is requested/released
node_store_check; % node at a state that indicate store access
  % is requested/released
store_store_check; % the environment to check access wrights
  % to a data store
env_store_check; % the environment is at a state where access
  % rights to a store can be checked
cdfd_cdfd_start; % cdfd process generated it's input data
  % diagram ready to execute
cdfd_start; % synchronised message to indicate that
  % the cdfd did receive data
cdfd_input: PortId; % indicate which input port of the cdfd
  % generated data

```

```

cdfd_output: PortId;% indicate which output port of the cdfd
  % consumed data
%consume_port: FSet(FlowId);
cdfd_consume;
node_flow: FlowId; % process node, ready to generate data
node_flow_consume: FlowId; % process node, read to consume data
node_flow_query: FSet(FlowId); % process node, to query data of
env_cdfd_start; % process environment to start with cdfd receiving
  % input data
env_flow_query: FSet(FlowId); % process env query the status
  % of flows containing data
env_flow_consume: FlowId; % process flow ready to for data to
  % be consumed
env_store_update: StoreId # Rights; % process env store to update
  % the access status
env_flow: FlowId;
env_node_start: NodeId; % environment indicate a node to
  % start execution
env_node_end: NodeId; % environment process wait for node
  % to complete execution
env_new_phase: Int # Phase;
store_update: StoreId # Rights;
node_store_update: StoreId # Rights;
node_execute: NodeId; % indicate a node is ready to execute
node_end: NodeId;
done_fire:NodeId;
flow_consume: FlowId; % a synchronised action to indicate
  % that data is consumed
flow_action: FlowId; % a synchronised action to indicate
  % that data is generated
flow_query: FSet(FlowId); % a synchronised action to
  % indicate that
  % the flow containing data is being queried
fire: NodeId;
fire_ports: NodeId # PortId # PortId; % indicate that a
  % node, input and output port fire
error;
skip;
start_fire: NodeId; % a synchronised message that indicate
  % when a node fire
selected_port_generate: NodeId # PortId; % select an output
  % port of a node that will generate data
env_move_to_end; % debug message to indicate the environment
  % move to its EnvironmentEnd state
env_wait_to_start:NodeId; % environment wait for a specific
  % node to start
env_new_start; % environment is ready for a new set of node
  % to fire
phase_process; % env move to state where process are allowed

```



```

% to execute
wait_env_store_check: Int;
exec_phase_process: List(NodeId); % process env, start the
% execution of process nodes
env_start_done; % process env done starting nodes to fire
store_update_to_process: List(StoreId); % process node, to
% update access to data stores
node_select_in_port: PortId; % node select inport that will
% consume data
node_done_access_update; % indicate node is done updating
% access to data store
error_r_w: Int # Int; % indicate an data store access violation

proc
NodeStoreAccess(ndid: NodeId, read_store_list: List(StoreId),
    write_store_list: List(StoreId)) =
    store_update_to_process(write_store_list).
    store_update_to_process(read_store_list).
    (#read_store_list > 0) ->
        node_store_update(head(read_store_list),
            read_access_begin).
        store_update_to_process(tail(read_store_list)).
        NodeStoreAccess(ndid, tail(read_store_list),
            write_store_list)
    <> (#write_store_list > 0) ->
        node_store_update(head(write_store_list), write_access_begin).
        store_update_to_process(tail(write_store_list)).
        NodeStoreAccess(ndid, read_store_list, tail(write_store_list))
    <>
    node_done_access_update.node_store_check.
    (sum fset: FSet(FlowId).node_flow_query(fset).sum ipid: PortId.
        (ipid in get_list_of_fire_ports(ndid, fset)) ->
            node_select_in_port(ipid).
            NodeFlowIdConsume(ndid,
                inPortDataFlowId(ndid, ipid), ipid)
    );

NodeStoreRelease(ndid: NodeId,
    read_store_list: List(StoreId),
    write_store_list: List(StoreId)) =
    (#read_store_list > 0) ->
        node_store_update(head(read_store_list), read_access_done).
        NodeStoreRelease(ndid,
            tail(read_store_list), write_store_list)
    <>
    (#write_store_list > 0) ->
        node_store_update(head(write_store_list), write_access_done).
        NodeStoreRelease(ndid, read_store_list, tail(write_store_list))

```

```

<>
node_end(ndid).node_store_check.Node(ndid);

NodeFlowIdConsume(
    ndid:NodeId, f_set :
    FSet(FlowId), pid:PortId) =
(f_set == {}) ->
    skip.NodeGenerate(ndid, pid)
<>
    sum f: FlowId . (f in f_set) ->
        node_flow_consume(f).
    NodeFlowIdConsume(ndid,
        f_set - {f}, pid);

NodeFlowIdGenerate(
    ndid:NodeId, f_set : FSet(FlowId),
    ipid:PortId, opid:PortId) =
(f_set == {}) ->
    fire_ports(ndid, ipid, opid).
NodeStoreRelease(ndid,
    node_read_list(ndid),
    node_write_list(ndid))
<>
    sum f: FlowId . (f in f_set) -> node_flow(f).
    NodeFlowIdGenerate(ndid, f_set - {f}, ipid, opid);

NodeGenerate(ndid:NodeId, ipid:PortId) =
    sum opidg:PortId.(opidg in io_relation(ndid, ipid)) ->
        selected_port_generate(ndid, opidg).
    NodeFlowIdGenerate(ndid,
        outPortDataFlowId(ndid, opidg),
        ipid, opidg);

Node(ndid:NodeId) =
    node_execute(ndid).fire(ndid).
    NodeStoreAccess(ndid, node_read_list(ndid),
        node_write_list(ndid));

CdfdIO =
    sum ipid: PortId.(ipid in cdfd_inputPort) ->
        cdfd_input(ipid).
        CdfdGenerate(ipid, cdfdInputFlowIds(ipid));

CdfdGenerate(ipid: PortId, flow_set:FSet(FlowId)) =
    (flow_set == {}) ->
        cdfd_cdfd_start.CdfdConsumeStart(ipid)
<>
    (sum f: FlowId. (f in flow_set) -> node_flow(f)).

```

```

CdfdGenerate(ipid, flow_set - {f}));

CdfdConsumeStart(ipid: PortId) =
  sum f_set: FSet(FlowId). node_flow_query(f_set).
  (f_set == {}) -> skip.CdfdConsumeStart(ipid) <>
  sum opid: PortId. (opid in cdfd_outputPort) -> (
    (f_set * cdfdOutputFlowIds(opid) ==
     cdfdOutputFlowIds(opid) &&
     cdfdOutputFlowIds(opid) != {}) ->
    cdfd_consume.
    CdfdConsume(ipid, opid, cdfdOutputFlowIds(opid))
    <>
    skip.CdfdConsumeStart(ipid)
  ) <> skip.CdfdConsumeStart(ipid);

CdfdConsume(ipid: PortId, opid: PortId, flow_set: FSet(FlowId)) =
  (flow_set == {}) ->
  cdfd_output(opid).CdfdConsumeStart(ipid)
  <>
  sum f: FlowId. (f in flow_set) -> node_flow_consume(f).
  CdfdConsume(ipid, opid, flow_set - {f});

EnvironmentFirst = env_cdfd_start.env_new_start.
  Environment(exec_control, 0);

Environment(phase: Phase, counter: Nat) =
  (counter < 2) ->
  env_new_phase(counter, phase). sum f_set: FSet(FlowId).
  node_flow_query(f_set). (
    (#get_fire_subset(getNodeOfType(phase), f_set, phase) > 0) ->
    exec_phase_process(getNodeOfType(phase)).
    EnvironmentStart(get_fire_subset(get_fire_subset(
    getNodeOfType(phase),
    f_set, phase), f_set, phase), [], phase, 0)
    <>
    phase_process.Environment(flip_phase(phase), counter + 1)
  )
  <>
  skip;

EnvironmentStartCheck(
  to_wait_set: List(NodeId),
  phase: Phase, w_count: Int) =
  (w_count == 0) -> env_move_to_end.
  EnvironmentEnd(to_wait_set, phase)
  <> wait_env_store_check(w_count).env_store_check.
  EnvironmentStartCheck(to_wait_set, phase, w_count - 1);

```

```

EnvironmentStart(nd_set:List(NodeId),
    to_wait_set:List(NodeId), phase:Phase, w_count:Int) =
(#nd_set == 0) ->
    env_start_done.EnvironmentStartCheck(to_wait_set,
        phase, w_count)
<>
    env_wait_to_start(head(nd_set)).env_node_start(head(nd_set)).
        EnvironmentStart(tail(nd_set),
            to_wait_set ++ [head(nd_set)],
            phase, w_count + 1);

EnvironmentEnd(to_wait_set:List(NodeId), phase:Phase) =
(#to_wait_set > 0) -> (
    env_node_end(head(to_wait_set)).
    env_store_check.EnvironmentEnd(tail(to_wait_set), phase))
<> (
    (phase == exec_process) ->
        env_new_start.Environment(exec_control,0)
    <> (phase == exec_control) ->
        env_new_start.Environment(exec_process,0)
);

EnvironmentFlowIds(f_set:FSet(FlowId)) =
    sum f_act: FlowIdAction.(
        (f_act == flow_consume) ->
            (sum c_FlowId:FlowId.(c_FlowId in f_set) ->
                env_flow_consume(c_FlowId).EnvironmentFlowIds(
                    f_set - {c_FlowId}))
        <> (f_act == flow_query) ->
            env_flow_query(f_set).
            EnvironmentFlowIds(f_set)
        <> (f_act == flow_generate) -> (sum c_FlowId:FlowId.!(
            c_FlowId in f_set) ->
            env_flow(c_FlowId).
            EnvironmentFlowIds({c_FlowId} + f_set))
);

EnvironmentStoreConstrain(read_map:StoreMap,
    write_map:StoreMap, sid:StoreId) =
(write_map(sid) > 1) ->
    error_r_w(read_map(sid), write_map(sid))
<> (write_map(sid) > 0 && read_map(sid) > 0) ->
    error_r_w(read_map(sid), write_map(sid))
<> (write_map(sid) < 0) ->
    error_r_w(read_map(sid), write_map(sid))
<> (read_map(sid) < 0) ->
    error_r_w(read_map(sid), write_map(sid))
<>

```

```

skip.EnvironmentStore(read_map, write_map);

EnvironmentStore(read_map:StoreMap, write_map:StoreMap) =
sum rid:Rights.
    (rid == write_access_begin) ->
        sum ssid:StoreId.env_store_update(ssid, rid).
EnvironmentStoreConstrain(read_map,
    update_access_count(ssid, write_map, 1),
    ssid)
    <> (rid == write_access_done) ->
        sum ssid:StoreId.env_store_update(ssid, rid).
EnvironmentStoreConstrain(read_map,
    update_access_count(ssid, write_map, -1),
    ssid)
    <> (rid == read_access_begin) ->
        sum ssid:StoreId.env_store_update(ssid, rid).
EnvironmentStoreConstrain(
    update_access_count(ssid, read_map, 1), write_map,
    ssid)
    <> (rid == read_access_done) ->
        sum ssid:StoreId.env_store_update(ssid, rid).
EnvironmentStoreConstrain(
    update_access_count(ssid, read_map, -1),
    write_map, ssid)
    <> store_store_check.EnvironmentStore(
    read_map, write_map);

init
allow({
    error, phase_process, store_check, start_fire,
    selected_port_generate, fire, fire_ports, flow_action,
    flow_query, flow_consume, skip, cdfd_input,
    cdfd_output, done_fire, cdfd_start, cdfd_consume,
    env_new_phase, env_new_start, env_wait_to_start,
    env_move_to_end, env_start_done, store_update,
    wait_env_store_check, store_update_to_process,
    node_select_in_port, node_done_access_update,
    error_r_w, exec_phase_process
},
comm({
    env_store_check | store_store_check | node_store_check
        -> store_check,
    env_cdfd_start | cdfd_cdfd_start -> cdfd_start,
    env_flow | node_flow -> flow_action,
    env_flow_consume | node_flow_consume -> flow_consume,
    env_flow_query | node_flow_query -> flow_query,
    env_node_start | node_execute -> start_fire,
    env_store_update | node_store_update -> store_update,
    env_node_end | node_end -> done_fire

```

```

    %env_cdfd_consume | cdfd_cdfd_consume -> cdfd_consume
  },
Node(NIBroadCastNode) || Node(NIConditionNode) || Node(NIP_cont) ||
Node(NIP_first) || Node(NIP_last) || Node(NIP_loop) ||
Node(NIP_noloop) || Node(NIP_second) || EnvironmentFirst ||
EnvironmentFlowIds({}) || CdfdIO ||
EnvironmentStore(init_store, init_store)
));

```

Listing B.3: The translation to mCRL2 of the small example

B.1.3 Results

CDFD	Refinement down for input port 1 <ul style="list-style-type: none"> • All conditions are satisfied Refinement down for input port 2 <ul style="list-style-type: none"> • All condition are satisfied Refinement up for output port 1 <ul style="list-style-type: none"> • All conditions are satisfied Refinement up for output port 2 <ul style="list-style-type: none"> • All conditions are satisfied Initialisation and invariant <ul style="list-style-type: none"> • All conditions are satisfied
Process P_{cont}	Pre image input port 1 all output ports <ul style="list-style-type: none"> • sat • unsat Pre image input port 1 output port 1 <ul style="list-style-type: none"> • sat • unsat Pre image input port 2 all output ports <ul style="list-style-type: none"> • All conditions are satisfied Pre image input port 2 output port 1 <ul style="list-style-type: none"> • All conditions are satisfied Pre image input port 3 all output ports <ul style="list-style-type: none"> • sat • unsat Pre image input port 3 output port 1 <ul style="list-style-type: none"> • sat • unsat Transition from input port 1 to output port 1 <ul style="list-style-type: none"> • sat • sat • unsat Transition from input port 2 to output port 1 <ul style="list-style-type: none"> • All conditions are satisfied Transition from input port 3 to output port 1 <ul style="list-style-type: none"> • All conditions are satisfied

Process P_{first}	<p>Pre image input port 1 all output ports</p> <ul style="list-style-type: none"> • sat • unsat <p>Pre image input port 1 output port 1</p> <ul style="list-style-type: none"> • sat • unsat <p>Transition from input port 1 to output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied
Process $P_{nolooop}$	<p>Pre image input port 1 all output ports</p> <ul style="list-style-type: none"> • sat • unsat <p>Pre image input port 1 output port 1</p> <ul style="list-style-type: none"> • sat • unsat <p>Pre image input port 2 all output ports</p> <ul style="list-style-type: none"> • All conditions are satisfied <p>Pre image input port 2 output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied
	<p>Transition from input port 1 to output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied <p>Transition from input port 2 to output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied
	<p>Transition from input port 1 to output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied <p>Transition from input port 2 to output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied
Process P_{last}	<p>Pre image input port 1 all output ports</p> <ul style="list-style-type: none"> • All conditions are satisfied <p>Pre image input port 1 output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied
	<p>Transition from input port 0 to output port 0</p> <ul style="list-style-type: none"> • All conditions are satisfied
Process P_{second}	<p>Pre image input port 1 all output ports</p> <ul style="list-style-type: none"> • All conditions are satisfied <p>Pre image input port 1 output port 1</p> <ul style="list-style-type: none"> • All conditions are satisfied
	<p>Transition from input port 0 to output port 0</p> <ul style="list-style-type: none"> • All conditions are satisfied
Process P_{loop}	<p>Pre image input port 1 all output ports</p> <ul style="list-style-type: none"> • sat • unsat <p>Pre image input port 1 output port 1</p> <ul style="list-style-type: none"> • sat • unsat <p>Pre image input port 1 output port 2</p> <ul style="list-style-type: none"> • sat • unsat <p>Pre image input port 2 all output ports</p> <ul style="list-style-type: none"> • All conditions are satisfied

Condition node	Pre image input port 2 output port 1
	• All conditions are satisfied
	Pre image input port 2 output port 2
	• All conditions are satisfied
	Transition from input port 1 to output port 1
	• All conditions are satisfied
	Transition from input port 1 to output port 2
	• All conditions are satisfied
	Transition from input port 2 to output port 1
	• All conditions are satisfied
Transition from input port 2 to output port 2	
• All conditions are satisfied	
Prove output predicates are disjunct	
• All conditions are satisfied	
Transition from input port 1 to output port 1	
• All conditions are satisfied	
Transition from input port 1 to output port 2	
• All conditions are satisfied	

Table B.1: Analysis of SMT-LIB results

Test if all input ports allow execution of the diagram	
Port 1	true
Port 2	true
Test a node can consume from more than input port in a fire set	
Node_BroadCastNode	false
Node_ConditionNode	false
Node_cont	false
Node_P_first	false
Node_P_last	false
Node_P_loop	false
Node_P_noloop	false
Node_P_second	false
Test if input ports allow generation of data on multiple outputs	
Port 1	true
Port 1	true
Port 2	true
Port 2	true
Test if processes can execute at least once	
Node_BroadCastNode	true
Node_ConditionNode	true
Node_P_cont	true
Node_P_first	true
Node_P_last	true
Node_P_loop	true
Node_P_noloop	true

Node_P_second	true
Test if processes can execute infinite number of time	
Node_BroadCastNode	false
Node_Cond	false
Node_P_cont	true
Node_P_first	false
Node_P_last	false
Node_P_loop	true
Node_P_noloop	false
Node_P_second	false
Test if the input port of the CDFD, can generate data on output port	
Port 1 generate results on Port 1	true
Port 1 generate results on Port 2	false
Port 2 generate results on Port 1	true
Port 2 generate results on Port 2	true
Test if each execution generate output data	
For every execution path starting with Port 1 generate data	true
For every execution path starting with Port 2 generate data	true

Table B.2: Results from evaluating μ -Calculus formulæ

B.2 Multi-port Example

```

module SYSTEM_Multi_in
  behaviour Parent_CDFD(ind:int) out_other:nat0
    state_condition in 1 out 1 pre true post true
    broadcast Cast(ind) in_1, alt end_broadcast
    flow behaviour.ind => cast Cast.ind
    flow cast Cast.in_1 => proc Process.in_1
    flow cast Cast.alt => proc Process.alt
    flow proc Process.out_other => behaviour.out_other
  end_behaviour
  process Init() end_process
  process Process(in_1:int| alt:int) out_other:nat0
    state_condition in 1 out 1 pre true post true
    state_condition in 2 out 1 pre true post true
  end_process
end_module

```

Listing B.4: The specification of the multi-port example

B.3 Producer

```

module SYSTEM_Delivery
  type
  Order = composed of
    product_type: int

```

```

    amount: int
end;
ProduceOrder = composed of Order and
  order_number:int
end;
Invoice = composed of
  cost: real
  order: Order
end;
ProductProduced = composed of
  //product: int
  order_number: int
  order: Order
end;
Payment = composed of
  //product: int
  money: real
  order: Order
end;
DeliveryVoucher = composed of
  product_reference: int
  order: Order
end;
OrderTrack = composed of
  order: Order
  order_number: int
  in_production: bool
  was_billed: bool
  ready_for_delivery: bool
  payment_received: bool
  reference_number: int
end

var
  orderTrack: set of OrderTrack

inv
  forall[o:OrderTrack | (
    (o inset orderTrack) (
      o.in_production or
      o.was_billed or
      o.ready_for_delivery or
      o.payment_received)
    )]

behaviour CDFD(order:Order|payment:Payment|voucher: DeliveryVoucher)
  invoice:Invoice|voucher_present:DeliveryVoucher, deliver:ProductProduced
  state_condition in 1 out 1 pre true post true

```

```

state_condition in 1 out 2 pre true post true
state_condition in 1 out 3 pre true post true
state_condition in 2 out 1 pre true post true
state_condition in 2 out 2 pre true post true
state_condition in 2 out 3 pre true post true
state_condition in 3 out 1 pre true post true
state_condition in 3 out 2 pre true post true
state_condition in 3 out 3 pre true post true

flow behaviour.order => proc Supervisor.order
flow behaviour.payment => proc Supervisor.payment
flow behaviour.voucher_present => proc Store.voucher_present

flow proc Supervisor.invoice => behaviour.invoice

flow proc Supervisor.product_order => proc Production.product_order
flow proc Production.product => proc Store.product
flow proc Supervisor.delivery_allowed => proc Store.delivery_allowed
flow proc Supervisor.voucher => behaviour.voucher
flow proc Store.item => behaviour.item

shadow_flow proc Production.product_produced =>
    proc Supervisor.product_produced
shadow_flow proc Store.store_acknowledge =>
    proc Supervisor.store_acknowledge
end_behaviour

process Init() end_process

process Supervisor(connect store_acknowledge|order:Order|
    payment:Payment|connect product_produced)
    product_order:ProduceOrder, invoice:Invoice|
    voucher: DeliveryVoucher,delivery_allowed:DeliveryVoucher

ext wr orderTrack

state_condition in 1 out 1 pre true post true
state_condition in 1 out 2 pre true post true

state_condition in 2 out 1 pre order.amount > 0
    post product_order.amount = ~order.amount and
    product_order.product_type = ~order.product_type and
    orderTrack = union(~orderTrack,
        {mk_ OrderTrack(
            ~order, true, false, false, false)}) and
    invoice.order = ~order
state_condition in 2 out 2 pre false post false

state_condition in 3 out 1 pre false post false

```

```

state_condition in 3 out 2
  pre payment.money > 0 and payment.order inset orderTrack
  post delivery_allowed.order = ~payment.order

state_condition in 4 out 1 pre true post true
state_condition in 4 out 2 pre true post true

comment "The Supervisor:
  - Receive an order from a client, production needs to start
  "
end_process

process Production(product_order:ProduceOrder)
  connect product_produced, product:ProductProduced
state_condition in 1 out 1 pre product_order.amount > 0
  post ~product_order = product.order and
    ~product_order.order_number = product.order_number

comment "The Production
  - Production starts when the supervisor inform the producer
  - When the product was product the supervisor is informed
  "
end_process

process Store(product:ProductProduced| voucher_present: DeliveryVoucher,
  delivery_allowed:DeliveryVoucher)
  item:ProductProduced|connect store_acknowledge
state_condition in 1 out 1 pre false post false
state_condition in 1 out 2 pre product.order.amount > 0.0 post true
state_condition in 2 out 1 pre voucher_present = delivery_allowed
  post ~voucher_present = item and
    voucher_present.product_reference = item.order_number
state_condition in 2 out 2 pre false post false
comment "The Store
  - Receive a product when produced and deliver when requested
  "
end_process

function Find_order(order: Order, order_data_list:set of OrderTrack)
  order_ret: OrderTrack
  /* scope of sets or lists are not allowed */
pre exists[
  o:OrderTrack |
  o inset order_data_list and order = o.order
]
  post order_ret.order = ~order
end_function

end_module

```