User Guide

Generated by Doxygen 1.8.4

Thu Apr 11 2019 16:01:26

Contents

Chapter 1

Pop user guide

This document is a user manual for the Pop Platform, a component of the Polychrony Toolset (see next figure).

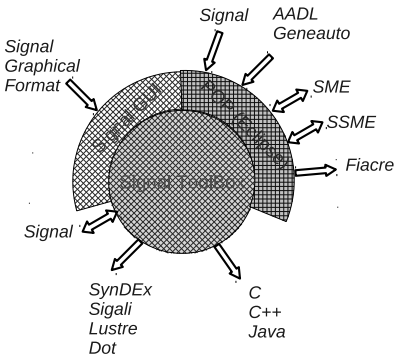

Figure 1.1: Polychony/Pop environment

It is a front-end to the **Signal** ToolBox in the Eclipse environment. It explains how to produce **Signal** or **SSME** (**SSME** stands for **Signal** Syntactic Model under Eclipse) models, transform them, verify them, and generate codes for simulations or for other tools. Morevover, it exists translators that produce **SSME** or **Signal** code from other formalisms. For more details consult the **Polychrony** Toolset web site.

Attention

To use the functionalities of the Signal ToolBox, the Signal ToolBox is supposed to be available in the Eclipse session:

- the Signal tool box is installed in the user environment. To install it, consult the Polychrony site and select the waited version.
- the setup of the Signal tool-box has been called before the Eclipse POP Session.

Pop constitutes a development environment for critical systems, from abstract specification until deployment on distributed systems. It relies on the application of formal methods, allowed by the representation of a system, at the different steps of its development, in the **Signal** polychronous semantic model.

The user will find here, many publications on Signal language (introductory papers, formal semantics), synchronous languages, case studies, domain application (Architecture and OS modeling, Avionics applications)...

The user guide is composed of the following sections:

- Creation of a Signal/SSME model
- Compiling Signal/SSME model
- Pop configuration

Note that the user can install the **Signal GUI** to edit its models. This plugin is available on the **Polychrony Toolset web site.**.

1.1 Creation of a new project

If you have no project in your workspace, you will need to create a new one:

- **Right-click** on the navigator view
- select **New ->** **Project ...**
- Select then a **General project** , this will be sufficient to model, and click on **Next ->** :
- Give a name to the project and click on **Finish**.

1.2 Creation of a Signal/SSME model

1.2.1 Creation of a Signal textual model

- **Right-click** on the project where you want to create your model,
- select **New->** **Other...**
- select **General ->** **File**,
- Click on the **Next** button,
- ***GIVE A NAME SUFFIXED BY SIG*** to the File name.
- Click on the **Finish** button

Then the created filed is opened in the Eclipse editor window.

1.2.2 Editing a Signal textual model

The environment integrates a simple text editor to manipulate **Signal** Text files under Eclipse. The current version provides only syntax highlighting for **Signal** keywords, for comments, and for constant value using primitive types (string, character, or numerical value).

To use this editor, you have only to **Double-click** on the file with extension **.SIG**. Maybe the **Signal** Text editor is not selected by default, so, you have to **Right-click** on the file and select **Open With->Signal Text Editor**.

1.2.3 Creation of a SSME model

Note that there is not a graphical user interface for the **SSME** format. It is used as output format for translators from other formalisms (AADL to **SSME**, Geneauto/Simulink to **SSME**,...). But, if a user want to create by hand a **SSME** model, the following method may be used.

To help the user during the creation of a new **SSME** model file, the reflexive editor has a wizard. To start the wizard,

- **Right-click** on the project where you want to create your model,

- select **New->** **Other...**
- select **SSME ->** **SSME Model** line. Once you have selected the **SSME** Model, the **SSME** wizard is opened:
  - choose the name of the diagram
  - Select **Next->** and then select the kind of the root for the model file. There are three kinds of root model elements:
    - List Module: it corresponds to a **Signal** library of components, types, and constants.
    - Process Model: it corresponds to a **Signal** component (process, node, action, function).
    - Operator Model: (misc) it corresponds to a **Signal** operator definition.
  - Click on **Finish** button. Then the new model is opened in the Eclipse editor window.

1.2.4 Editing a SSME model

To define your model, you only have to add new model objects. To do so, **right-click** on the node on which you want to add a child, and select the **New Child** menu. It displays the list of all possible model elements that can be added for the current selected element. If this option does not exist for a node, it means that there is no possible child for this node.

1.3 Compiling Signal/SSME models

To compile a **Signal** or **SSME** program,

- **Select** a file (suffixed by **SSME** or SIG)
- **Rigth click** and **select the Polychrony Service** in the menu, then select one of the compiling modes:
  - **Batch compiling** (for more details, consult the Signal Batch Compiler Options documentation)
  - **Execute a predefined scenario** (for more details, consult the predefined scenarios documentation)
  - **Interactive compiling** (for more details, consult the Interactive compiling documentation)

1.4 Pop configuration

The **Signal** compiler can be parameterized. To access to these parameters, select the **Polychrony** section in the **Preferences Window**. You can

- add search paths for the **Signal** libraries
- modify some internal size values.

Chapter 2

Signal Batch Compiler Options

2.1 General rules:

- A simple option is a string -XXX preceded by a ' ' character; ' ' and '=' do not occur in XXX
- A valued option is a string -XXX = YYY preceded by a ' ' character, where
  - = can be preceded or not, followed or not by a ' ' character
  - YYY cannot start with a '-' character if it is preceded by a ' '; ' ' does not occur in YYY
- The name of an option XXX is either only made of capital letters, or only made of lower case letters
- If an option does not have a capital letter name, this option is transformed into lower case letters
- An option that has a mix of lower and upper case letters is transformed into lower case letters
- Unless othewise specified the ordering of independent arguments is free and meaningless FILE argument must have one and only one occurrence in the command line.
- The value of an optional cumulative argument is the upper bound of its occurrences.
- A non cumulative argument cannot have several occurrences.
- Let WD be the absolute path of the current directory, unless otherwise specified a path file FILE refers to
  - PATH=path(WD,FILE)=WD/FILE file or directory if FILE is relative,
  - PATH=path(WD,FILE)=FILE file or directory if FILE is absolute.
- Unless othewise specified the upper bound of an action notation induces the processing of the action

2.2 Options Controlling the Input

- [-in =] FILE where FILE denotes the Signal source file path(WD,FILE).
- -inisf = FILE where FILE denotes the Intermediate Signal Format Source file path(WD,FILE).

A module should have parameters (coming soon)

- -par=PARF PATH=path(WD,PARF) contains the Signal static parameter list for instantiating a process FOO
- -par (coming soon)
  - is equivalent to -par=FOO.PAR when the compiled file contains a Signal process FOO
  - is equivalent to -par=FOOi.PAR foreach exported process of the Signal module contained in the compiled

2.3 Options Controlling the Output

Initial syntax errors are written to the standard error file.

- -[d,D]=DIR output files are created in directory
  - PATH=path(path(WD,DIR),FOO), if d is present
  - PATH=path(WD,DIR) if D is present
  - PATH=path(WD,FOO) if none of these marks is present. this option cannot have more than one occurrence
- -war (case insensitive) displays warnings on standard output, or in FOO_LIS.SIG file if -lis is present
- -v (case insensitive)(verbose) display detailed log to standard output
- -tm (case insensitive) enables the traceability mode
- -print=w (case insensitive) unary sampling expression is written as [:H] instead of when H in generated TRA files.

2.4 Options Controlling the kind of output files

- -outisf = FILE creates a Signal file PATH/FOO.isf containing the Intermediate Signal Format of FOO definition; context errors (such as type,...)
- -lis, (case insensitive) create a Signal file PATH/FOO_LIS.SIG containing the pretty printed FOO definition; context errors (such as type,...) are not listed to get those errors -basic is required
- -tra create a Signal process file PATH/FOO_XXX_TRA.SIG containing the result of inter format translation
  - XXX = BASIC elementary clock reductions
  - XXX = POLY (BASIC) clock static resolution (maximal triangularisation)
  - XXX = ENDO (POLY) endochronous parametrization (single master clock)
  - XXX = BOOL (ENDO) event to Boolean clock transformation
  - XXX = SCH equation sequencing
- -TRA create a Signal process file for every inter format translation as -tra option does and every other action: the result of the action YY in the format XXX is unparsed in PATH/FOO_XXX_YY_TRA.SIG
- -sme (case insensitive) generates a file FOO.sme containing the SIGNAL meta-model of FOO.
- -ssme (case insensitive) generates a file FOO.ssme containing the SIGNAL Syntactic meta-model of FOO.
- -spec (case insensitive) generates a file FOO_ABSTRACT.SIG containing the interface abstraction of FOO
- -ec[=i] (case insensitive) generates a file FOO_EXCLUSIVE_CLOCKS.SIG containing the exclusive clocks pairs as descibed below
  - -ec is equivalent to -ec=0
  - -ec=0 only the clocks of the signals are selected
  - -ec=1 the selected clocks are the clocks of the signals and the clocks referred to in the expressions
  - -ec=2 all the clocks are selected
- -header (case insensitive) restricted to module file (M): generates a module that contains headers of processes defined in M

2.5 Options Controlling the Kind of export file

- -z3z, (case insensitive) export the control in z3z format FOO.z3z file containing the SIGALI abstraction of FOO FOO_POLY_TRA.SIG
- -lustre, (case insensitive) create NOTHING FOO.sdx file containing SynDEx target code
- -c[=[i][m]] (case insensitive)
  - -c: creates FOO.c, FOO.h, and files of FOO_ext.h, FOO_undef.log, FOO_type.h, that are needed in FOO.c
  - -c=i creates FOO_io.c and files of FOO_undef.log, FOO_type.h, that are needed in FOO_io.c
  - -c=m creates FOO_main.c
  - -c=im, -c=mi creates files created by -c=i and -c=m.

  c creates files created by c= and c=im where (ANSI C code)
  - FOO_externalsProc.h: created if some external functions are referred to in FOO contains interface of those functions
  - FOO_externalsUNDEF_LIS.h: created if some referred to types or constants are used and not defined in FOO; in this case FOO_externalsUNDEF.h must be provided
  - FOO_types.h: contains C types corresponding to Signal types
  - FOO_body.c: contains code associated with each step and the scheduler
  - FOO_body.h: contains interface of functions generated in FOO_body.c
  - FOO_io.c: contains input output functions associated with interface of FOO
  - FOO_main.c: contains main C program.
- -threads (associated with C option) in this case a thread is generated for each cluster and the scheduling is managed using semaphores.
- -c++[=[i][m]] (case insensitive) same as -c[=[i][m]] with C++ code
- -java [=[i][m]] (case insensitive) mostly the same as -c[=[i][m]] with java code
  - FOO.java:contains code associated with each step and the scheduler
  - FOO_io.java: contains input output functions associated with interface of FOO
  - FOO_main.java: contains main Java program
- -syndex, (case insensitive) create
- -force, (case insensitive) forces the code generation by adding “exceptions” for constraints
- -check, (case insensitive) generates code for assertions
- -umd, (case insensitive) the values are copied from the delayed variable to the original one (for non scalar types, the default is to manage the delayed signal and the original one in a circular area)
- -udo, (case insensitive) the delayed signal and the original one are in independent variables (the default is to manage the delayed signal and the original one in the same memory when it is possible)

2.6 Options Controlling the graph transformations

- -dr, (case insensitive) UPWARD normalization of delayed signals
- -crew=[b][d] (case insensitive) b: event normalization of Boolean clock expressions d: when/default definitions are rewritten (operand reduced to a signal)
- -su (case insensitive) signal syntactic equivalence reduction

- -ds (case insensitive) replaces the default definitions by partial definitions (default splitting)
- -pdg (case insensitive) replaces partial definitions by default expressions (partial definitions grouping)
- -s=n, (case insensitive) variable substitution n is an integer definition is substituted to each signal that has at most n occurrences
- -kernel (case insensitive) split expressions
- -uc (case insensitive) reduce the clocks of the signal upto their utility clock
- -profiling generates a file containing a signal processus that computes the execution time of the original one

2.7 Partition Options

- -sso (case insensitive) state variables isolation
- -sbo (case insensitive) Boolean nodes isolation
- -sfto (case insensitive) separate finite and not finite types
- -clu (case insensitive) code separation wrt input predecessors equivalence
- -dist (case insensitive) user defined code separation

2.8 Interformat translations:

For all these options, when the capital letter style is used, the resulting XXX graph is unparsed in a file FOO_XXX-_TRA.SIG

- -basic, -BASIC in case of context errors (such as type,...) a file PATH/FOO_LIS.SIG with error annotations is generated
- -poly[=n], -POLY[=n]
- -depth=n (case insensitive) tuning of Boolean reduction algorithm (for wizards only), DEPRECATED use -poly=n n is an integer (by default, n=5)
- -bC DEPRECATED use -bool
- -bool, -BOOL (for Boolean Control) produces the SIGNAL code without "events"; the hierarchy of clocks is completed by Boolean inputs
- -fl DEPRECATED use -flat
- -flat, -FLAT produces the SIGNAL code in which the hierarchy of clocks is reduced (flattened) to one level
- -sD DEPRECATED use -seq
- -seq, -SEQ produces a SIGNAL sequential code

(∗) Level formats:

- BASIC the control is explicit but unsolved
- POLY the control is a forest of trees of event clocks, each tree is hierarchized
- ENDO the control is a forest reduced to one tree of event clocks, the tree is hierarchized
- BOOL the control is a forest reduced to one tree of Boolean clocks, the tree is hierarchized
- FLAT the control is a forest reduced to one tree of Boolean clocks, the tree is flattened
- SCH BOOL level in which the scheduling (static or partial) is explicit

Chapter 3

Interactive compiling

This mode allow to a user to apply interactively functionalities on its **Signal** program. When it is activated, it proposes a set of applicable functionalities (see figure below) according to the internal representation of the program: the compiling of a **Signal** program is a set of **Signal** program transformations, but some transformations may be applied depending on the previously applied transformations.

For example, to generate the C code, many steps are required, the program must be:

- syntactically correct,
- well formed (no double declarations,...),
- well typed,
- without clock constraints (by default),
- without definition cycles,....

So the set of applicable transformations evoluates during the compiling.

Currently, a restricted set of functionalities are proposed. Note that, these functionalities are the same than those proposed for the definition of a compilation scenario.

Chapter 4

Some predefined scenarios

To call **Polychrony** services, right-click on the **SSME** file on which you want to apply the service(s), and select **Polychrony**. You can choose to use a predefined scenario : in this case, select the Execute Predefined Scenarios option.

Seven actions are proposed, from the selected **SSME** or (textual) **Signal** model:

- Generate textual **Signal** file (SIG) for a **SSME** model, or **SSME** file for a textual **Signal** model.
- Generate C files (equivalent to the application of the options -c -TRA -forced of the batch compiler)
- Generate C++ files (equivalent to the application of the options -c++ -TRA -forced of the batch compiler)
- Generate Java files (equivalent to the application of the options -java -TRA -forced of the batch compiler)
- Generate SIG (LIS) file from the selected model (equivalent to the application of the option -lis of the batch compiler)
- Generate SIG (TRA) file from the selected model (equivalent to the application of the option -tra of the batch compiler)
- Execute a compilation scenario on the model

The code C/C++/Java is generated if the model is well formed, well typed, endochronous, without cycles, and without clock constraints. All the generated files are generated in a directory named as the **SSME** or **Signal** model file. They are only generated if all previous compilation steps are all successful.

Chapter 5

a subset of the Polychrony functionalities

These functionalities are proposed in interactive compiling mode and for the definition of a scenario compilation. Note that these functionalities are also available in batch mode compiling using options (given in parenthesis).

Transformations

- **Retiming** (-dr): performs an UPWARD normalization of delayed signals. It rewrites synchronous function f such that Y := f(X1 $ m1 init V1,...,Xn $ mn init Vm) into Y := y' $ j init f(V1',...,Vm') and y' := f(X1 $ m1' init V1",...., Xm $ mn' init Vm").
- **Booleans to events** (-crew): performs an event normalization of boolean clock expressions (example: when (a and b) -> when a when b).
- **Signal unifications** (-su): performs a signal syntactic equivalence reduction. For example, for (| x:=E | y :=E |) x is replaced by y.
- **Clock calculus** (-poly): performs the resolution of the clock systems using a triangularization technique. The result is a forest of clock trees (hierarchy with several roots).
- **Endochronisation** (-endo): reduces the hierarchy with several roots (polychronous model) to one root (endochronous model).
- **Events to booleans** (-bool) : produces an endochronous model without "events".
- **Sequential clustering** (-clu) : performs a code separation with respect to input predecessors equivalence.
- **Abstraction** (-spec): computes the abstraction of the program (I/O data dependences, I/O clock relations, the "black Box" or the "grey Box" abstraction representation).
- **Sequentializing** (-seq): produces a SIGNAL sequential code (reinforcing of the dependencies)
- **Flattening** (-flat): produces the SIGNAL code in which the hierarchy of clocks is reduced (flattened) to one level.

Export tools

- **Sigali**: Generates code (.z3z files) for `Sigali tool`, used to prove dynamical properties.
- **Lustre**: Generates lustre code (.lus file).
- **Syndex**: Generates code (.sdx file) for `SynDEx tool`, used for code distribution.

Code generators

- **C ANSI**: Generates C code (.c and .h files), with clusters if the **Sequential clustering** ha been applied.
- **C++**: Generates C++ code (.cpp and .h files), with clusters if the **Sequential clustering** has been applied.

- **Java**: Generates Java code (.java) with clusters if the **Sequential clustering** has been applied.

Export Signal

- **Signal Textual**: produces the textual Signal file for a SSME model.
- **Signal Model (SSME)**: produces the SSME form for a textual Signal file.
- **Signal Textual (LIS)**: produces a Signal file containing the pretty printed definition.
- **Signal Textual (TRA)**: produces a Signal file containing the result of the applied transformations.
- **Signal Abstraction**: produces a Signal file containing the abstraction of the compiled model (I/O clock relations, I/O dependences, ...).
- **Profiling**: produces a morphism of the compiled program for profiling. The path of the morphism table must be in the SIGNAL_LIBRARY_PATH shell variable.

Chapter 6

Compilation scenario

The **SSME** Scenario View (see following picture) constitutes a way to describe a compilation scenario with some assistance. Each functionality and generator is represented by a button and, according to the functionality or generator you activate, others become available or are disabled.

The reflexive editor has been automatically generated from the compilation scenario meta-model. To create a new compilation scenario file (**.ssc**),

- **Right-click** on your project
- **Select New->Other...** and then select the following model : **Polychrony->SSME Compilation Model**.

However, there are some constraints to create a compilation scenario, because some functionalities/generators can only be applied after others, so an interactive view (described in next part) has been created to help user to create such scenario.

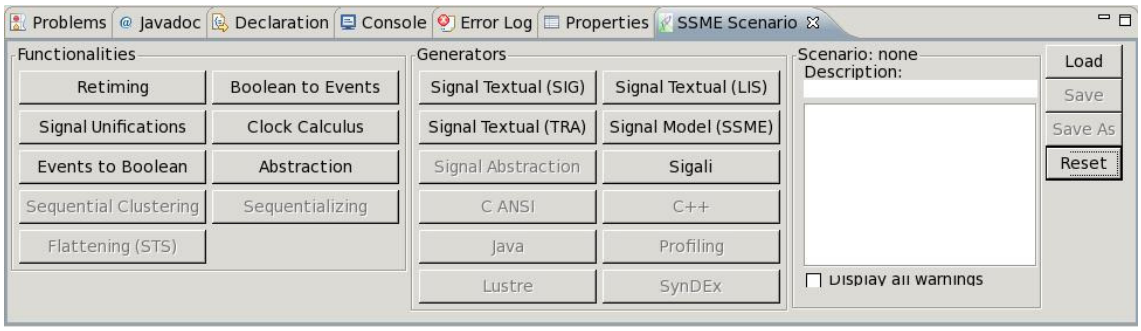

Figure 6.1: Scenario view

To access to this view,

- select **Window-> Show View-> Other...**,
- select **Polychrony->SSME Scenario**