Automatic Code Generation from Stateflow Models

Andres Toom

IB Krates OÜ / Institute of Cybernetics at TUT

Based on the Master’s Thesis 05.2007
Supervisors: Tõnu Näks, Tarmo Uustalu
TUT Department of Computer Control
and the Gene-Auto Project

Theory Days, Vanaõue 2007

Outline

- Introduction
  - The Gene-Auto Project
  - Model Based System Design
    - Declarative style
    - Imperative style
- Stateflow
  - Informal introduction
  - Modelling considerations
- Formal specification of Stateflow
- Code generation from Stateflow
- Demo
- Conclusions

Introduction

The Gene-Auto project

ITEA 2

Paris / November 21st 2006

ITCT 2006

Gene-Auto Consortium

Partners
- Industrial "users"
- Services "suppliers"
- SME's
- Research Institutes
- Universities

Duration : 3 years from January 2006

Countries : Belgium, Estonia, France, Israel

ITEA project, Aerospace Valley (ISAURE)

The Gene-Auto project (contd.)

- Motivations
  - Increasing complexity of embedded real-time systems
  - Increasing demands for safety and reliability
  - Shorter time-to-market development pressure
  - Existing closed proprietary systems lack in flexibility and their vendors deny any liability for using their products.
- Aims
  - Develop an open source code generator from mathematical style systems modelling languages (e.g Simulink/Scicos, Stateflow)
  - Full qualification of the code generator according to the industry standards
  - Integrate formal methods, as much as possible to reduce the amount of classical testing
  - Initial target language is (platform independent) C
- Current work
  - Provide a code generator prototype for the Stateflow language to explore and refine the functionality and semantics of Stateflow.

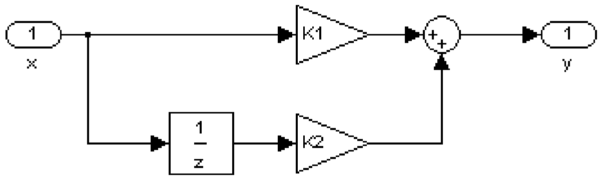
Specifying dynamic/reactive systems

- Two styles:
  - Declarative ~ data-flow
  - Imperative ~ automata
- Synchronous vs. asynchronous models
  - Synchronous:
    - Synchronicity hypotheses - computation instants are instantaneous and atomic, time passes only between the computations.
    - Simpler to handle.
    - Both, declarative and imperative variants exist:
      - Lustre, Signal, … – synchronous data-flow
      - Esterel, StateMate, … – synchronous automata
  - Asynchronous
    - Computations take time and are non-atomic
    - More general, more complex
  - GALS – Globally Asynchronous Locally Synchronous

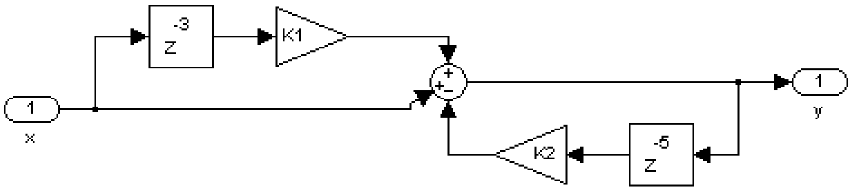
Declarative style of modelling dynamic/reactive systems

- Functional modelling, (mostly) data-flow oriented.
- Well suited for expressing systems represented as a set of differencial or difference equations.
- Examples:

$$y(n) = K_1x(n) + K_2x(n-1)$$



$$y(n) = x(n) + K_1x(n-m_1) - K_2y(n-m_2)$$



Declarative style of modelling dynamic/reactive systems (contd.)

- Many visual modelling tools exist
  - Simulink, Scicos, Scade (Lustre), Sildex (Signal), Polychrony (Signal), …
- Synchronous data-flow languages provide a rigorous formalism for specifying many systems
  - Operate on (infinite) sequences of values over time
  - Formal methods, e.g. model checking, can be applied on such models
  - See for example, N. Halbwachs EWSCS’06.
- Simulink
  - Most widely used mathematical modelling tool in practice
  - Background in modelling continuous systems
  - No rigorous formalism underneath. The semantics of the modelled system is defined by its behaviour during the simulation.
  - Complete semantics more complex and powerful than that of synchronous data-flow languages.

Imperative style of modelling

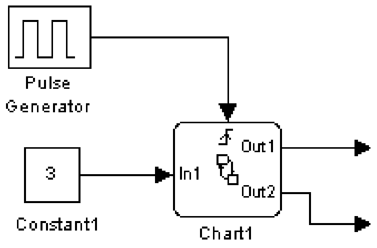
- Synchronous language Esterel
    - SyncCharts, Safe State Machines (SSM)
- Statecharts
    - A visual formalism for specifying the behaviour of dynamic systems.
    - Extends the classical finite state machine formalism, by adding:
        - depth (hierarchy)
        - orthogonality (parallel states)
        - broadcast communication.
    - Informal semantics proposed by David Harel in 1987 (1).
    - Formal semantics, called the Statemate semantics of Statecharts, presented in 1987 (2) and 1996 by D. Harel et al.
- By 1994 over 20 variants of Statecharts existed that tried to refine some aspect of it.

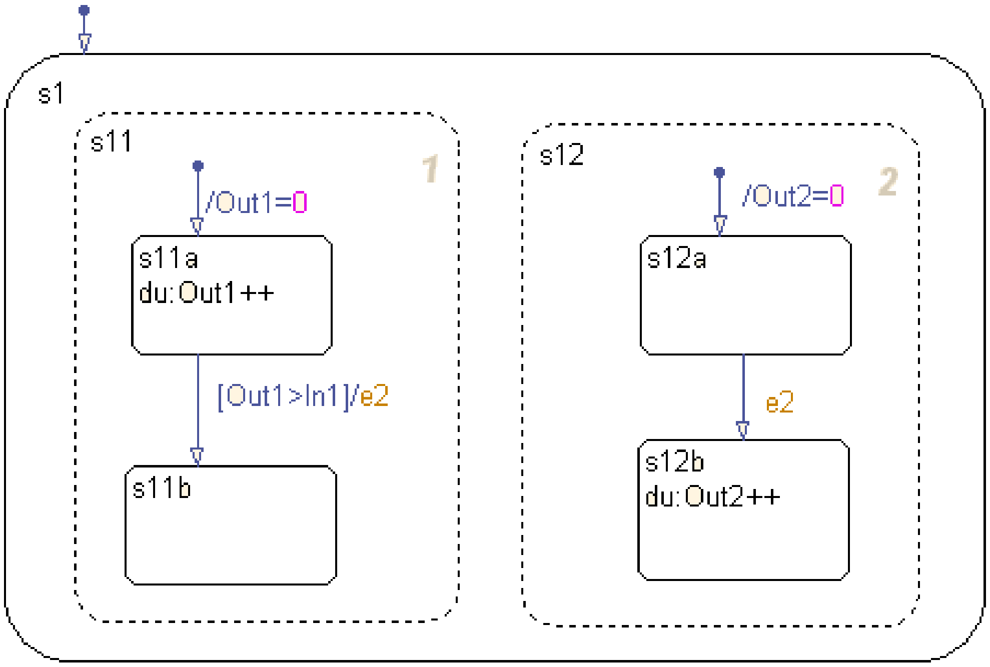
Stateflow

Stateflow

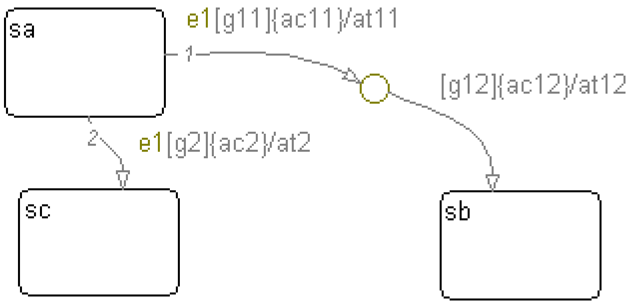
- Based on the Statecharts formalism.
- Designed by the Mathworks Inc, part of the Matlab/Simulink toolset.
- Several unique additions.
  - Combines StateCharts, flow-charts and truthtables in a unique way.
- A complex transition and action mechanism.
- Very expressive, but with caveats for the modeller.

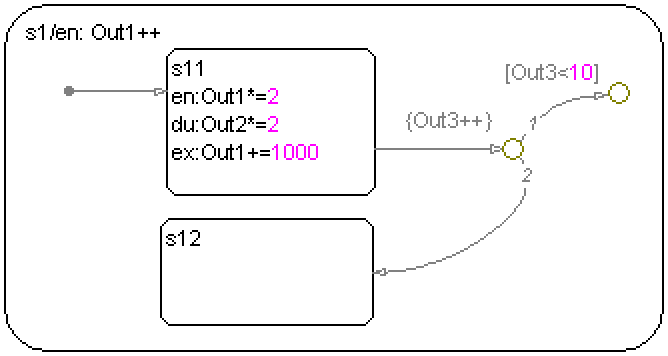
Simulink/Stateflow - Example

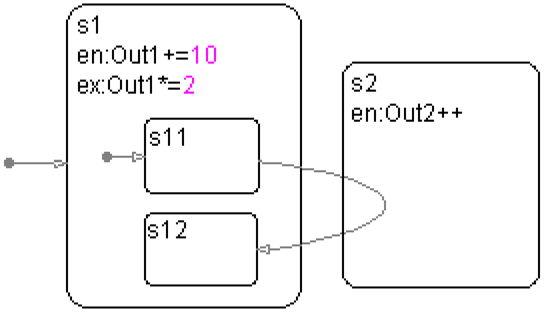


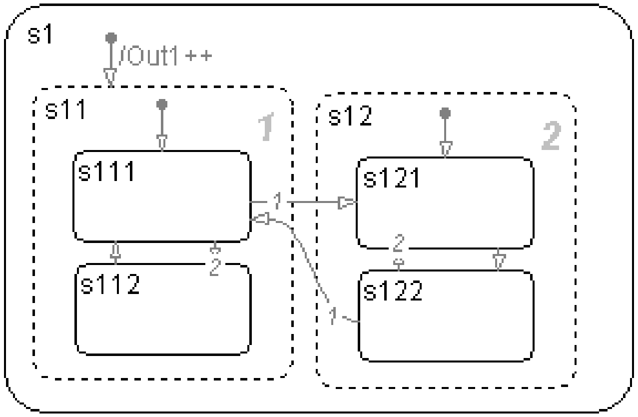


Stateflow – Modelling caveats
Puzzling semantics

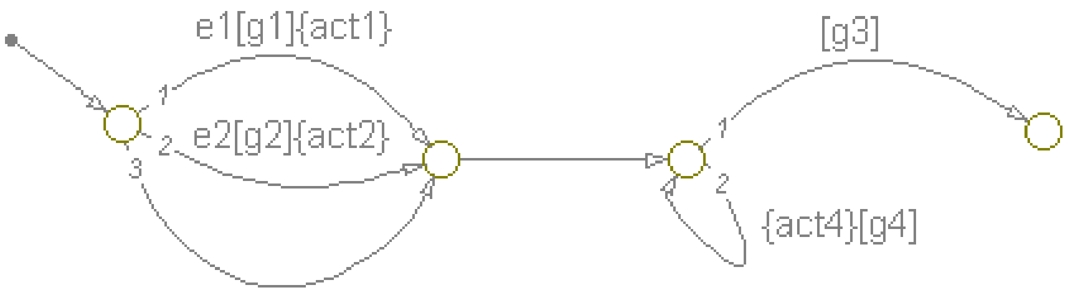


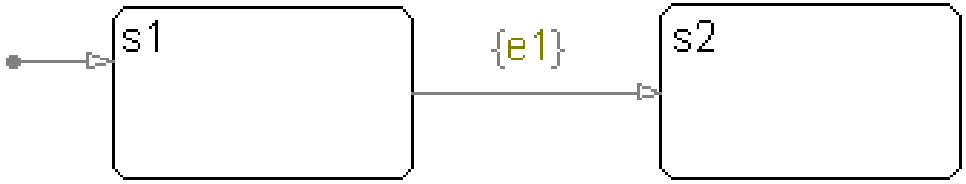






Stateflow – Modelling caveats (contd.)
Non-termination





Modelling restrictions!

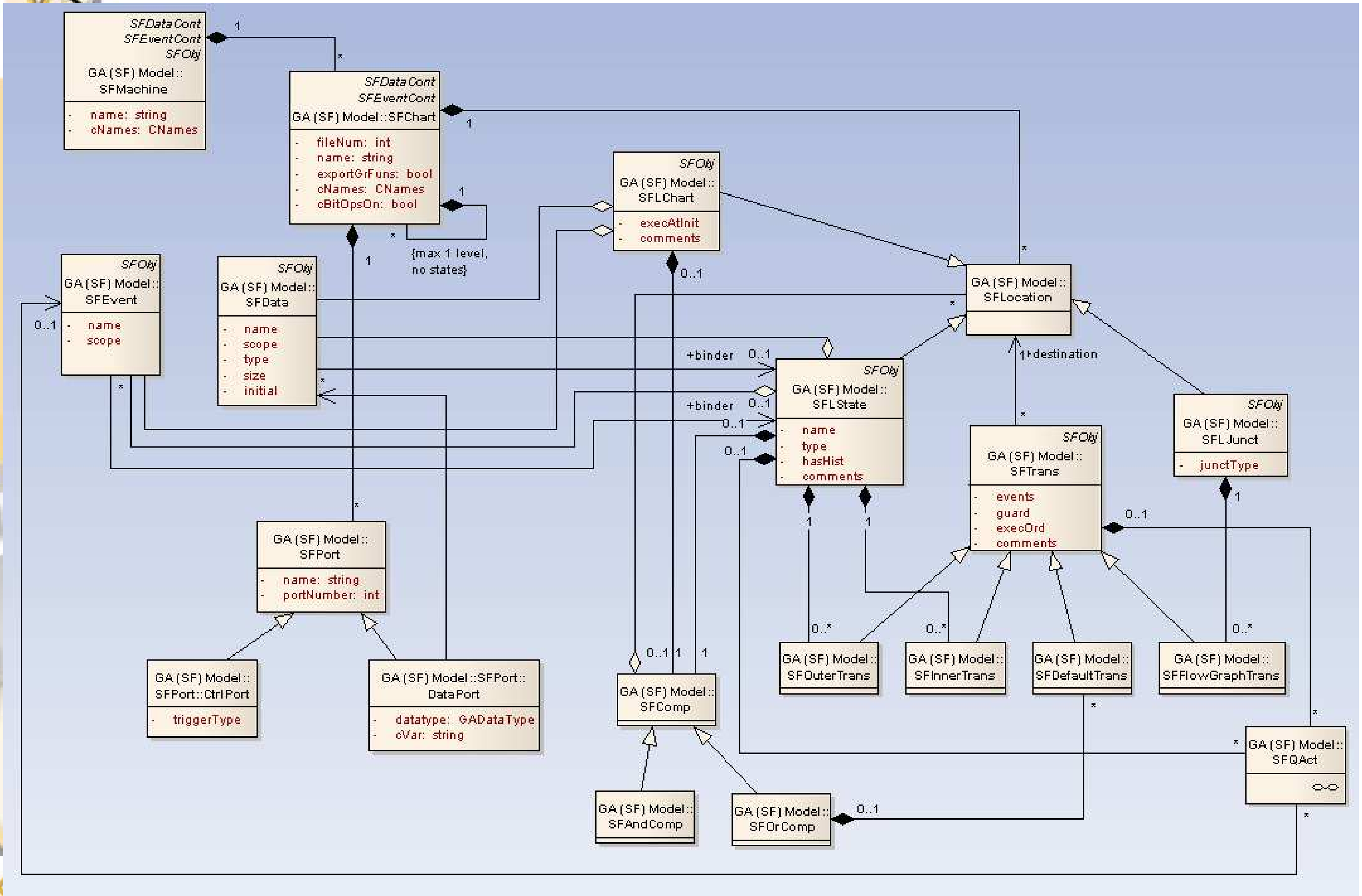
- Complex semantics can easily lead to misestimating the exact run-time behaviour.
- Possibilities for non-termination of the computation exist.
  - Such constructs are specifically forbidden in some other languages: e.g. Esterel, Safe Sate Machines.

Formal specification of Stateflow

The Stateflow Language

- Informally defined by the Mathworks.
  - Reference manual is over 900 pages.
  - The *de facto* semantics is defined by the simulation.
- Formal definition of a subset of Stateflow.
  - Operational semantics - G. Hamon and J. Rushby (2004).
  - Denotational semantics - G. Hamon (2005).

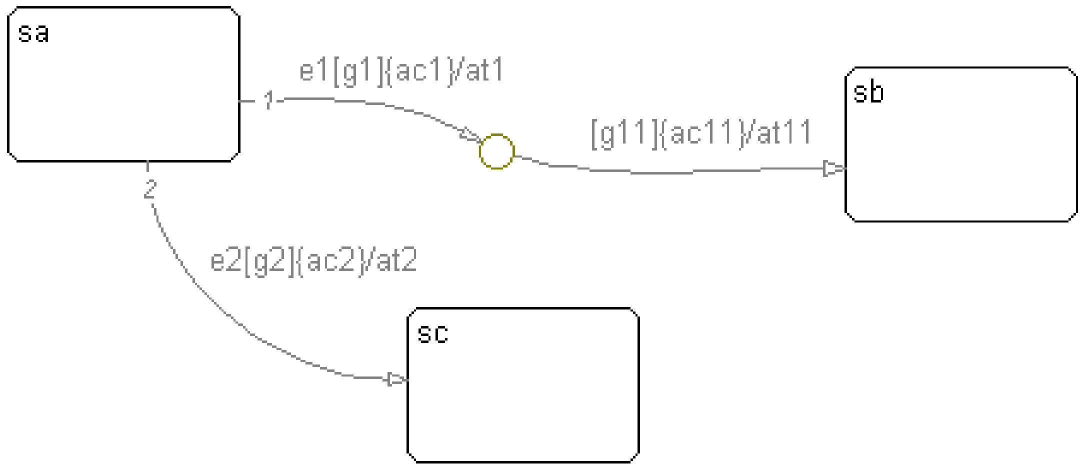
Stateflow syntax (Gene-Auto SF Metamodel)



Denotational semantics of Stateflow

- Approach from G. Hamon (2005)
- Environment
  - Contains bindings of *user variables* and chart's *statevariables* to values
    - `type Env = (Maybe (Array SFDataId SimVal), (Array SFLocId LocState)`
- Continuation environment
  - Not used in the current implementation
  - Defunctionalizing the continuation environment yields just `SFChart` – SF language semantics is kept separate from the input model's structure
- Continuations to express the transition semantics
  - Success:
    - `type Kplus  = Env → Dest → Env`
  - Failure:
    - `type Kminus = Env → Env`

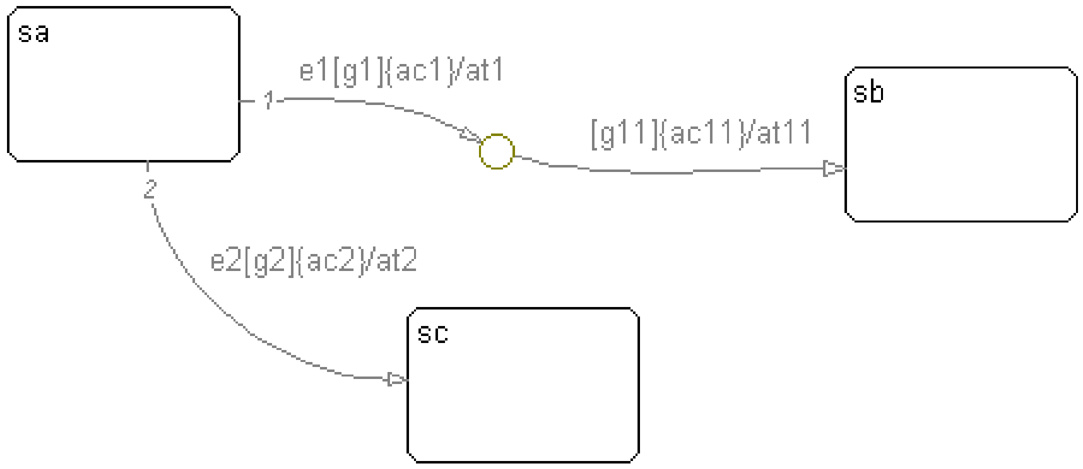
Success and fail continuations



- Continuations
  - A mathematical formalism, capable of handling full jumps in computer programs (i.e. “gotos”)
  - Intuition - a way to formally deal with the “rest of the program”

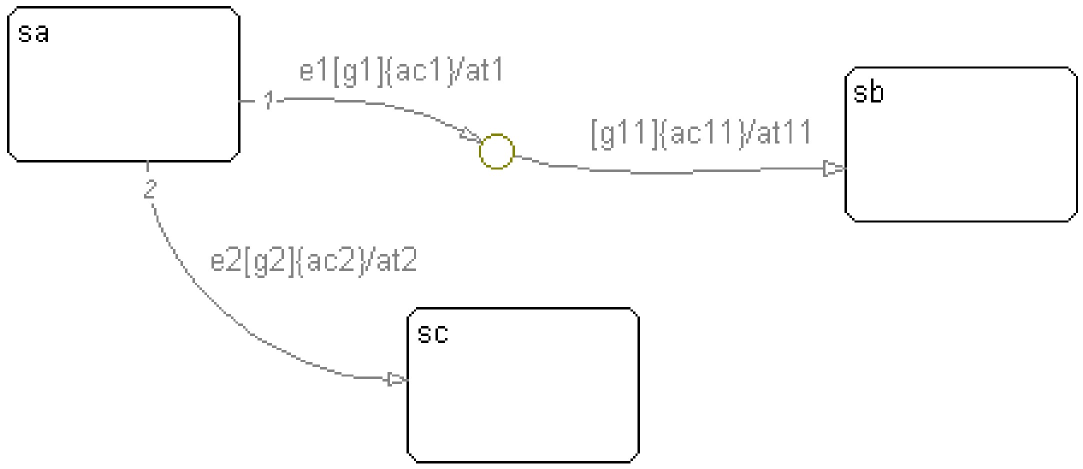
C. Strachey, C. P. Wadsworth

Success and fail continuations



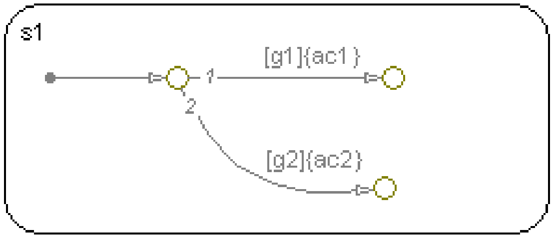
- Continuations to express the transition semantics
    - Success:
        - `type Kplus  = Env → Dest → Env`
    - Failure:
        - `type Kminus = Env → Env`

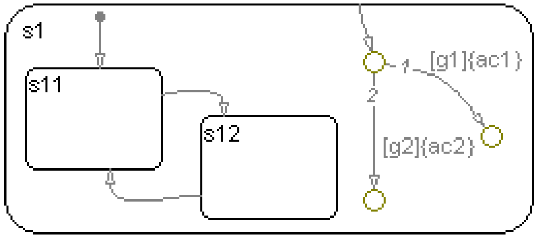
Revised success continuation

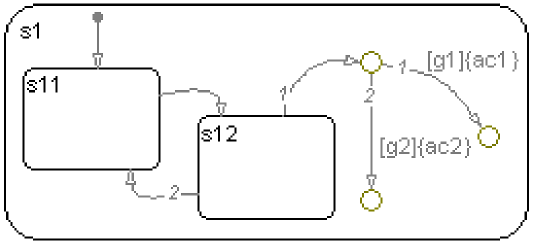


- Continuations of type: `Env` → `Dest` → `Env`
  - Are insufficient to correctly build the evaluation sequence of actions/activites
  - Need a different approach,

Revised success continuation (contd.)

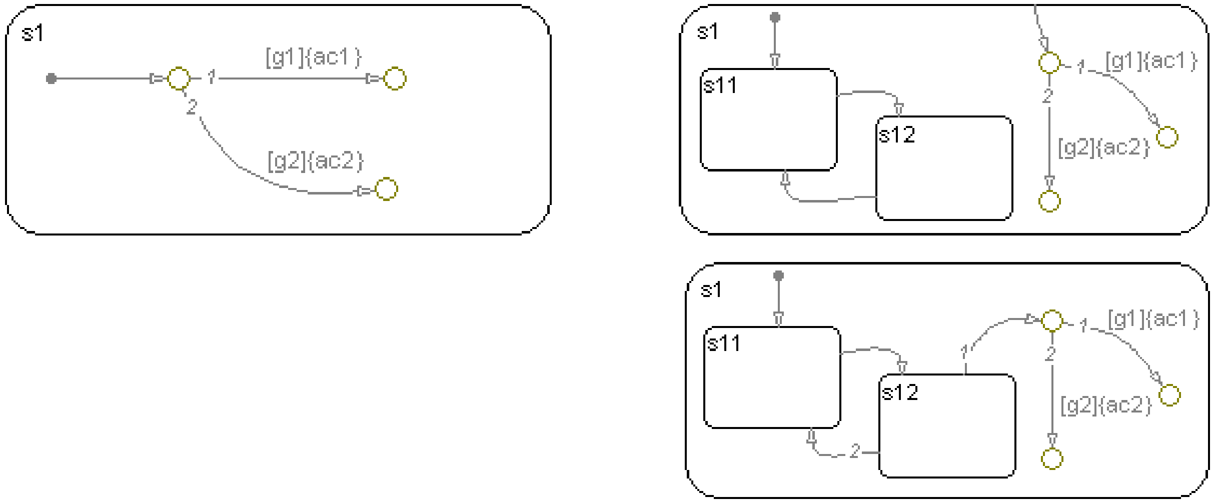






- Second problem:
  - What to do, when terminal junctions appear together with states?
- Need a third continuation type:
  - `type KTerm  = Env → Env`
- And
  - Distinguishing between pure flow-graph networks and flow-graphs networks mixed with states.

Revised success continuation (contd.)



- Revised success continuation type:
  -
    ```
    data KPos = KPosFG KTerm
          | KPosOuter SFStateId [SFAct] KTerm
          | KPosInner SFParentId [SFAct] Kterm
          | KPosDefault SFParentId [SFAct] KTerm
    ```
  - (Defunctionalized)

Semantical functions

- Evaluating a chart

```
runChart :: SFChart -> Env -> SFEventId -> Env
runChart k r e =
    if not envIsOpen r (chdChartId k) then chartEnter k r e
                                      else chartExec  k r e
```

- Entering a chart

```
chartEnter :: SFChart -> Env -> SFEventId -> Env
chartEnter k r e =
    let c = sfcGetChart k
    in let r'  = envOpenLoc k r (sfcChartId k)
       in compEnter (sfcChartId k) (chartGetComp c) k r' e
```

- Entering a composition …
- Entering a state …

Semantical functions (contd.)

- Evaluating a transition

```
evalTrans :: SFTrans -> ChartDef -> Env -> KPos -> KNeg ->
             SFEventId -> Env
evalTrans t k r success fail e =
    if (isValidEvent (transGetEvents t) e) `and` (checkGuard
             (transGetGuard t) r) then
        let success' = kposAddTransActs success
             (transGetTransActs t)
            r' = doActs (transGetCondActs t) k r
        in evalDest (transGetDest t) k r' success' fail e
    else
         fail r
```

Semantical functions (contd.)

- Evaluating a transition list

```
evalTransList :: [SFTrans] -> ChartDef -> Env -> KPos ->
                 KNeg -> SFEventId -> Env

evalTransList []     k r success fail e = fail r

evalTransList (t:ts) k r success fail e =

    let fail' = \rf -> evalTransList ts k rf success fail
                 e

    in evalTrans t k r success fail' e
```

Code generation from Stateflow

Code generation via partial evaluation of the semantics

- The semantic function for evaluating the chart:
  - `runChart :: SFChart → Env → EventId → Env`

- Result of partial evaluation against the `SFChart`:
  - `runChart' :: Env → EventId → Env`

Specific considerations with partial evaluation – Inlining amount

- Need a way to control evaluation.
- First, we don’t want to evaluate everything, because:
  - run-time computations must not get evaluated during the code generation
  - we might not want to give a specification of primitive functions
- One solution
  - Supply a list of abstract or primitive functions to the evaluator:
    - `[(Identifier, Arity)]`

Specific considerations with partial evaluation – Inlining amount (contd.)

- Second, a naive evaluation would inline still too much.
  - For example, consider following action statements:
    - `uData[1] = …`
    - `uData[2] = uData[1] + 1`
    - `uData[3] = uData[1] + 10`
  - A straightforward evaluation would recreate the evaluation sequence of `uData[1]` several times.
- Goal
  - Inline/evaluate only the “meta-actions” – actions related to evaluating the chart’s semantics.
  - Do not evaluate the actions that have intended effects on the environment.
- One solution
  - Augment the “specification language” with a special construct.

Specific considerations with partial evaluation – Inlining amount (contd.)

- A `local` “keyword” is introduced.
- Defined in Haskell as follows:

```
local :: a -> a
local    x =  x
```

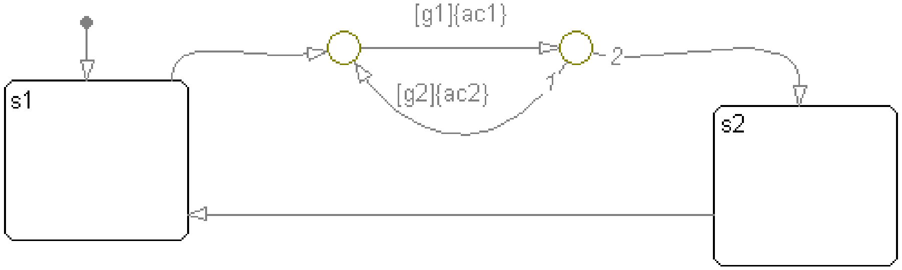
- Used as an indicator for the partial evaluator to preserve a local let-binding.
- Example:

```
let … = …
in let r'  = local envOpenLoc k r (sfcChartId k)
   in compEnter (sfcChartId k) (chartGetComp c) k r' e
```

- Automatic alternatives are possible.

Specific considerations with partial evaluation – Loops

- Loops in the evaluated program.
- Flow-graph loops in Stateflow correspond to loops in traditional imperative programs and in general may not terminate.



- The partial evaluator needs to a criterion to stop.

Specific considerations with partial evaluation – Loops (contd.)

- A `lbl` “keyword” is introduced.
- Defined in Haskell as follows:

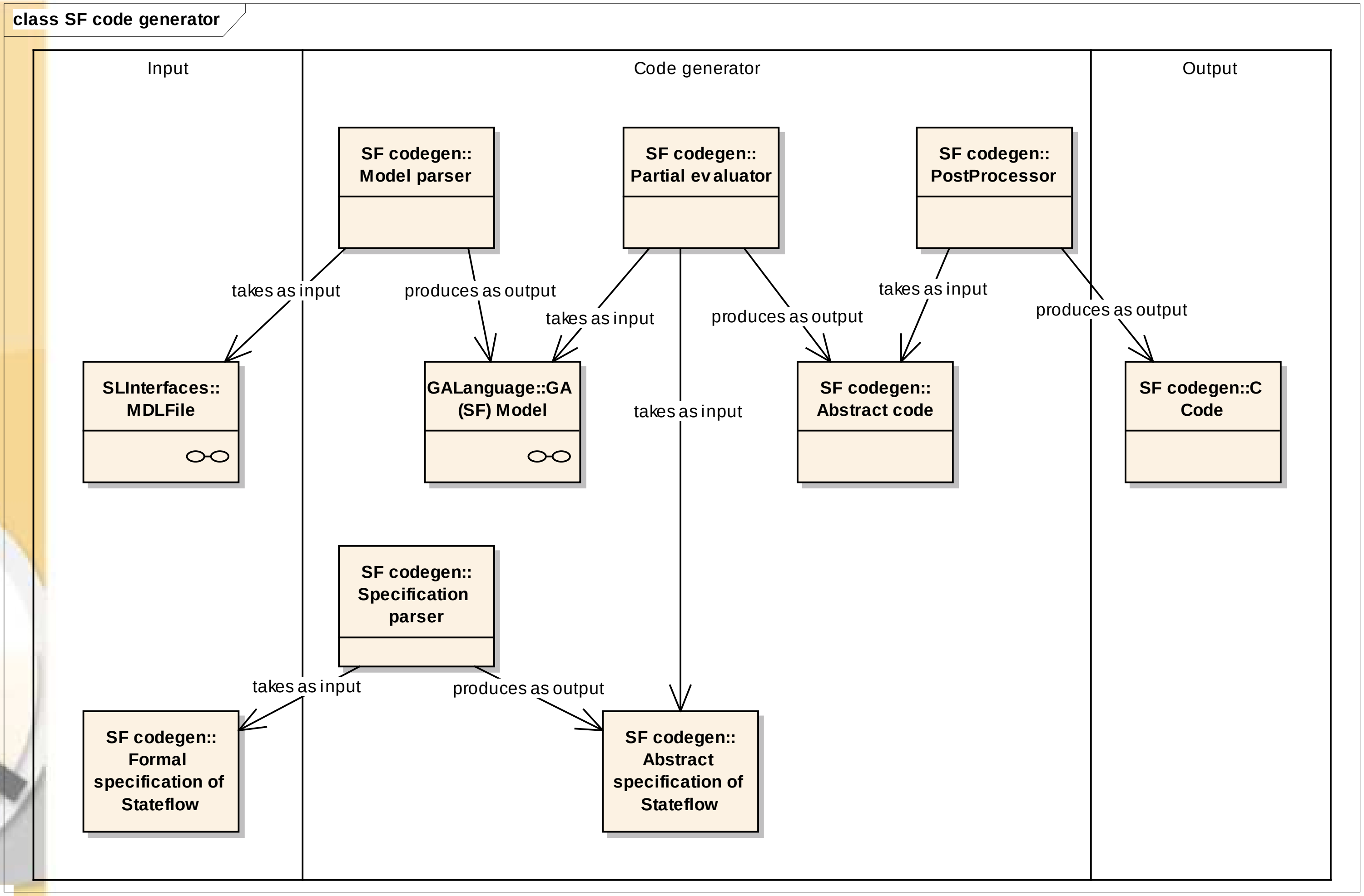
```
lbl :: Int -> a -> a
lbl    i      b =  b
```

- A special meaning for the partial evaluator:
  - a “label” has to be generated the first time a `lbl` with a new number is seen and a “goto” any other time. The rest of the expression is evaluated only the first time.
- In Stateflow this also solves the issue of evaluating joining paths.
- Certain jumps can be transformed to `while`s, `for`s or `if`s later.

The HTr partial evaluator

- Developed for the purpose of the current project.
- Input language Haskell. Supported constructs:
    - pattern matching
    - function and constructor application (incl. infix application)
    - lambda abstraction
    - if and case expressions
    - local let binding
    - tuple and list expressions and list construction
- Introduced additional “language constructs”
    - for specifiying primitives, maintaining locality and dealing with loops
- Generic, does not know Stateflow
- Implemented also in Haskell.

Tool architecture with a partial evaluator



Code generation via manual transformation of the semantics

- Main idea
  - Keep the form of the original executable specification.
  - Rewrite it so that instead of outputting a modified environment it will output an expression that does that.
- The semantic function for evaluating the chart:
  - `runChart  :: SFChart → Env → EventId → Env`
- Transformed function:
  - `runChart’ :: SFChart → CmStmt`

Manual transformation of the semantics. Example

- Original semantic function

```
evalTrans :: SFTrans -> SFChart -> Env -> KPos -> KNeg -> SFEventId -> Env

evalTrans t k r success fail e =
    if (isValidEvent (transGetEvents t) e) `and`
       (checkGuard (transGetGuard t) r) then
          let success' = kposAddTransActs success (transGetTransActs t)
              r' = local doActs (transGetCondActs t) k r
          in evalDest (transGetDest t) k r' success' fail e
    else
          fail r
```

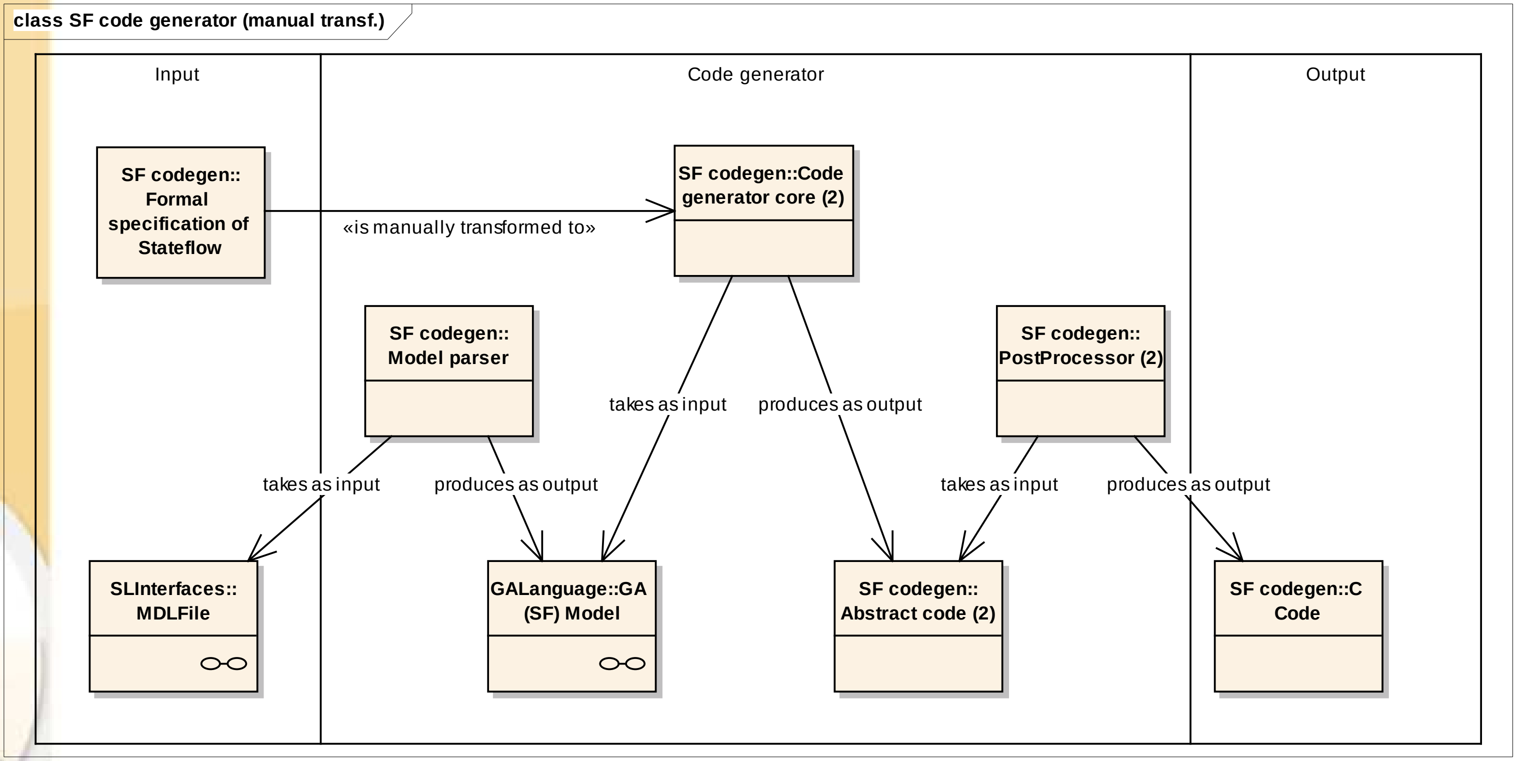
Manual transformation of the semantics. Example (contd.)

- Transformed semantic function

```
evalTrans :: SFTrans -> SFChart -> SeenLocs -> KPos -> KNeg -> (SeenLocs, CmStmt)

evalTrans t k ls success fail =
    let condExp = (isValidEvent (transGetEvents t)) `and`
                  (checkGuard (transGetGuard t))
        (ls', thenStmt) =
            let success' = kposAddTransActs success (transGetTransActs t)
            in seqStmts [doActs (transGetCondActs t),
                         evalDest (transGetDest t)  k success' fail] ls
        (ls'', elseStmt)  = fail ls'
    in (ls'', mkIf condExp thenStmt elseStmt)
```

Tool architecture with manually transformed semantics



Demo

Results

- A refined version of formal semantics of Stateflow has been specified.
- A code generator prototype based on that semantics has been created.
- Small-scale tests show conformance with the *de facto* Stateflow semantics.
- Some secondary features remain to be implemented to enable testing on real industrial test-cases.
- Creating a qualified version of the tool and using the results presented here in creating a formally validated code generator remain subjects for future work.