An Open Alternative for SMT-based Verification of SCADE Models

Henning Basold[1], Henning Günther[2], Michaela Huhn[3], and Stefan Milius[4]

[1] Radboud University Nijmegen and CWI Amsterdam, The Netherlands
h.basold@cs.ru.nl
[2] Institut für Informationssysteme, Technische Universität Wien, Austria
guenther@forsyte.at
[3] Department of Informatics, Clausthal University of Technology
Clausthal-Zellerfeld, Germany
Michaela.Huhn@tu-clausthal.de
[4] Lehrstuhl für Theoretische Informatik, FAU Erlangen-Nürnberg
Erlangen, Germany
mail@stefan-milius.eu

Abstract. SCADE is an industrial strength synchronous language and tool suite for the development of the software of safety-critical systems. It supports formal verification using the so-called Design Verifier. Here we start developing a freely available alternative to the Design Verifier intended to support the academic study of verification techniques tailored for SCADE programs. Inspired by work of Hagen and Tinelli on the SMT-based verification of LUSTRE programs, we develop an SMT-based verification method for SCADE programs. We introduce LAMA as an intermediate language into which SCADE programs can be translated and which easily can be transformed into SMT solver instances. We also present first experimental results of our approach using the SMT solver Z3.

1 Introduction

The software of safety-critical systems needs to fulfil strong requirements concerning its correctness. This is why great efforts are made to verify, validate and certify such software. A model-based development accompanied by formal verification is a well accepted means to frontload and complement quality assurance for software. In fact, formal methods, in particular formal verification techniques, are highly recommended by safety standards, such as DO-178B [11] for the avionics domain or EN50128 [8] for the railway domain, in a software process appropriate for the higher safety integrity levels.

For many safety-critical systems, synchronously clocked controllers are the preferred implementation method. SCADE[5] is an industrial strength modelling language and tool suite for the development of such controllers. Its language is based on the synchronous data flow language LUSTRE (Halbwachs et al. [17]) and was extended by

[5] SCADE is developed and distributed by Esterel Technologies, see www.esterel-technologies.com.

various features, most importantly, by so-called safe state machines (André [2]). The tool suite includes, among other features, code generation, graphical modelling, test automation, and the SCADE Design Verifier (DV) for SAT-based verification[6].

However, while SCADE DV performs very well for certain verification tasks, it can fail badly for others due to complexity problems. In the latter case the user has little to no information guiding to the causes making verification infeasible. This makes it almost impossible to assess whether it may be most promising to take further measures to make the formal verification task at hand eventually feasible, to settle for a weaker verification result (e.g. using bounded model checking using the *debug strategy* of SCADE DV) or even abandon further formal verification attempts and rather invest more efforts in testing. This is a disadvantage both for practical application in industry and for research pertaining to formal verification with SCADE DV (see, e.g., the study [20] on formal safety analysis of two industrial SCADE models). It may also explain the industries' indecision towards adopting formal verification in productive processes.

The work we present here is intended as a first step to counteract the above disadvantage. It is inspired by Hagen and Tinelli's work [15,16] on the SMT-based verification of synchronous LUSTRE programs. We build on their ideas to make the first steps towards a new verification method for SCADE models. After recalling necessary preliminaries in Sec. 2, we introduce the language LAMA in Sec. 3. This language is intended as an intermediate language into which synchronous models such as SCADE models can be translated and that allows for an easy generation of SMT instances. In the ensuing Sec. 4 and 5 we then describe the translation of SCADE synchronous programs to LAMA and of LAMA programs to SMT instances. Further, in Sec. 6 we present a prototypical implementation of a verification tool based on these translations. As an SMT solver we use Z3 [26]. We also provide first experimental results comparing our verification tool with SCADE DV. While SCADE DV still outperforms our tool, our experiments provide an argument for the correctness of our translations. In addition, our verification tool is freely available online. Hence, the LAMA language and our implementation can serve as a platform for the further academic study of the verification of synchronous SCADE programs, in particular for trying out various optimization and abstraction techniques in future work.

Related work. This paper reports the results of the first authors master's thesis [5]. We already mentioned Hagen's and Tinelli's work [15,16] on the SMT-based verification of LUSTRE programs resulting in the model checker KIND (using Yices [12] as SMT solver). Recent progress on this using parallelization was reported by Kahsai and Tinelli [22]. The basic ideas for verifying synchronous models using a SAT solver and induction go back to Sheeran et al. [28], and were implemented in the LUCIFER tool [23], a precursor of SCADE DV. The basics and usage of SCADE DV were reported by Abdulla et al. [1].

There are several methods and tools available for the formal verification of LUSTRE programs. The Lesar tool comes with the LUSTRE distribution [19,27]. NBAC [21] is a verification tool that is founded on abstract interpretation. Luke is a verification tool written by Koen Claessen which is an inductive verifier using an eager encoding into

[6] SCADE DV uses a SAT solver developed by Prover Technologies, see `www.prover.com`.

a SAT solver. Rantanplan by Franzén [14] is an incremental SMT-based verification tool for the inductive verification of LUSTRE programs; Franzén compared his tool with NBAC and Luke. Champion et al. [9] proposed to enhance k-induction based verification for LUSTRE by automated lemma generation. In the STUFF tool they joined property-directed heuristics and the arbitrary combination of system variables to come up with invariants that allow to strenghten the property to be proven.

An alternative approach to the verificaton of SCADE programs was developed at Rockwell Collins (see Whalen at al. [29]). This approach makes use of a transformation from SCADE to LUSTRE that was provided by the SCADE code generator at the time. LUSTRE programs are then translated into SAL ("Symbolic Analysis Laboratory", see [25]), which also uses Yices as SMT solver. This translation is not freely available but was reimplemented by Hagen and Tinelli [16] to compare performace with KIND; the latter outperformed the SAL based verification in most cases. The SAL language comes quite close to LAMA but is missing automata; it does have quantifiers (over values), though, a feature not present in LAMA. Unfortunately, the translation from SCADE to LUSTRE is no longer provided by current SCADE versions.

2 Preliminaries

We begin with a very brief overview of the SCADE language and formal verification using SCADE DV in Sec. 2.1; for a detailed language description see [13].

In Sec. 2.2–2.4 we give a brief overview over the role of Satisfiability Modulo Theories (SMT), the SMT-logic and background theories we are using, and how we are encoding the semantics of synchronous systems using this logic and theories.

2.1 SCADE and SCADE DV

As we mentioned already, SCADE is a mixture of a LUSTRE-based synchronous dataflow languages and so-called safe state machines. The basic building blocks of a SCADE model are the *operators*, each of which declares its input, output and local (state) variables. The type system supports simple datatypes (**bool**, **int**, ...) as well as enumerations, arrays and records. The LUSTRE like dataflow part of the language allows to connect inputs to outputs using (among others) logical and arithmetic operations, case distinction ("if-then-else") and up to iterators (map and fold) on arrays. Information can be stored in local variables across clock cycles and, importantly, the delay operators fby ("followed by") and pre allow to access values of variables from previous clock cycles. The safe state machines allow to switch control between different dataflow diagrams. They support hierarchy, i.e., states can contain arbitrary dataflow or state machine models. Safe state machines and dataflow models are fully integrated. Figure 1 shows an example SCADE node (without the interface specification).

The behavior of a SCADE model is formally captured as a transition system on which SAT-based model checking of safety properties can be performed [1]. SCADE DV does not offer a temporal logic but properties have to be modeled as combination of an invariant and *synchronous observers* [18] in the SCADE modeling language. Such an observer for a given SCADE operator is itself a SCADE operator that receives the

inputs and outputs of the operator to be observed and signals through a Boolean output whether the safety property it monitors holds. SCADE DV then verifies whether the Boolean output of the observer in parallel composition with the given SCADE operator is always **true**.

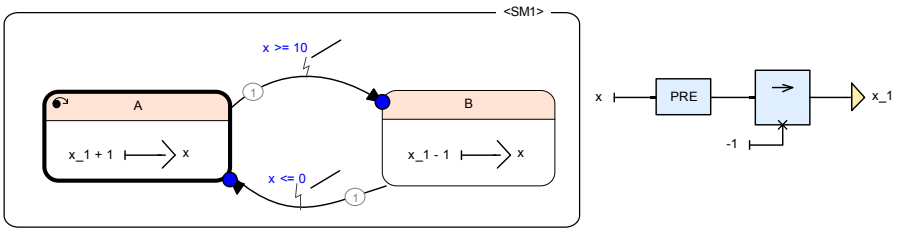

Fig. 1. Periodic counter in SCADE from [24]

2.2 Satisfiability Modulo Theories

In many cases performing model checking by fully exploring the state space of a system is infeasible at best, or even impossible. A lot of research has been devoted to tackle this problem. One possibility is to perform so-called *symbolic model checking* by encoding a system into logical formulas (usually in first-order logic). Most such descriptions use some so-called "background theory" $\mathcal{T}$ [3], which is independent of the system under consideration (e.g. integer arithmetic). It might be possible to encode such a theory $\mathcal{T}$ as "library" in first-order logic, but this can lead to performance problems. Usually, the required theories are nicely behaved, so a general purpose solver can be replaced by a specialized solver for $\mathcal{T}$. Using a fixed background theory $\mathcal{T}$, systems can often be described by quantifier free formulas. The validity of these formulas is then efficiently checked by combining a SAT-solver with a specialized solver for $\mathcal{T}$. This combination of SAT with specialized theories is called *Satisfiability Modulo Theories (SMT)*.

The SMT-solver that we are using in our implementation is Z3 [26]. To communicate with the solver we use the standardized text format SMT-LIBv2 [4].

2.3 Simplified SMT-Logic

In this paper we are using an instance of SMT with theories for arithmetic on the integers $\mathbb{Z}$, the rationals $\mathbb{Q}$ and the finite rings $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$. Moreover, we need to be able to make inductive definitions on the naturals $\mathbb{N}$ and combine these basic types using a product type. Besides the usual arithmetic operators and relations among the basic types we will use λ-abstraction over variables of the basic types above. To ease readability we favor a specialized syntax over SMT-LIBv2, even though SMT-LIBv2 is used in the implementation. The syntax we use is displayed in Fig. 2. The used variables x range over a set Var of term variables, the special symbols ite and p_1, p_2 are functions having the expected meaning of if-then-else and product projections, respectively. The arithmetic operations and relations are overloaded for $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{Z}_n$, and formulas can be used as terms of type $\mathbb{B}$ (Boolean).

$$
\begin{aligned}
\mathit{Terms} \ni t &::= x \mid c \mid \varphi \mid t \,\square\, t \mid \mathrm{ite}(\varphi, t, t) \mid \lambda x.t \mid t\ t \mid (t,t) \mid p_1(t) \mid p_2(t) \\
&\quad\ \square \in \{+, -, *, /\} \\
\mathit{Forms} \ni \varphi &::= x \mid \top \mid \bot \mid \neg\varphi \mid \varphi \,\square_1\, \varphi \mid t \,\square_2\, t \\
&\quad\ \square_1 \in \{\wedge, \vee, \rightarrow\} \quad \square_2 \in \{\equiv, <, >, \leq, \geq\}
\end{aligned}
$$

Fig. 2. Syntax of the used SMT terms and formulas

Another feature we require is the ability to define uninterpreted symbols. We write $\Sigma = v_1 : A_1, \ldots, v_n : A_n$ for the uninterpreted signature with variables v_i of type A_i. If Γ is a set of formulas, then we say that a formula φ is *valid* in Σ and Γ, if φ holds for any assignment to variables in Σ under the assumption of Γ, denoted as $\Sigma; \Gamma \vDash \varphi$.

2.4 Streams

One way of giving semantics to synchronous programs is by viewing them as stream transformations, i.e., functions taking streams of inputs to streams of outputs. A *stream* over a set X is a map $\sigma : \mathbb{N} \to X$. We denote the set of all streams over X by X^ω, hence a *stream transformation* is a map $X^\omega \to Y^\omega$.

The output of a synchronous program depends only on "what happened so far". More precisely, synchronous programs are in the class $\mathcal{C}(X, Y)$ of *causal* stream transformations, where $f : X^\omega \to Y^\omega$ is causal if for all $\sigma \in X^\omega$ and $n \in \mathbb{N}$ the value $f(\sigma)(n)$ only depends on $\sigma(0), \ldots, \sigma(n)$. The reason is that synchronous programs work step-wise, i.e., they are given by a map $c : S \times X \to S \times Y$ taking a state $s \in S$ and an input $x \in X$ to $c(s, x) = (s', y)$, a new state s' and an output y. Transition maps like c are known as *Mealy machines*, and inherently forbid "to look into the future".

We can represent streams over X directly in the SMT language from Sec. 2.2 as function symbols of type $\mathbb{N} \to X$. Assume we are given a transition map c, then the semantics of c for an input stream σ is given by the predicate

$$\mathsf{Iter}(c, \sigma, \gamma, \tau, n) := (\gamma(n+1) \equiv c_1(\gamma(n), \sigma(n))) \wedge (\tau(n) \equiv c_2(\gamma(n), \sigma(n))), \quad (1)$$

where $\gamma : S^\omega$ is the stream of internal states, $\tau : Y^\omega$ the stream of outputs and $c_i = p_i \circ c$, $i = 1, 2$. If σ, γ and τ are understood from the context, we just write $\widehat{c}_n = \mathsf{Iter}(c, \sigma, \gamma, \tau, n)$. Given an *initial condition* $s_0 : S$, the formula $\Delta_n = (\gamma(0) \equiv s_0) \wedge \bigwedge_{i=0}^n \widehat{c}_i$ defines $\gamma(i)$ and $\tau(i)$, $i = 0, \ldots, n$ uniquely, i.e., it approximates the streams γ and τ up to the n-th position. Just using τ, we can thus approximate the corresponding causal $f \in \mathcal{C}(X, Y)$: $\Delta_n \vDash f(\sigma)(n) \equiv \tau(n)$.

This approximation is the basis for bounded model checking: given a predicate $P(x, s, y)$ on $X \times S \times Y$, we can check that P holds up to depth n by showing $\Sigma; \Delta_n \vDash \bigwedge_{i=0}^n P_i$. Here we simplify again notation by writing $P_i = P(\sigma(i), \gamma(i), \tau(i))$ and, moreover, we use $\Sigma = \sigma : X^\omega, \gamma : S^\omega, \tau : Y^\omega$.

On the other hand, if we want to prove that the predicate P holds for every $n \in \mathbb{N}$, we can show this by induction, i.e., by showing that $\Sigma; \Delta_0 \vDash P_0$ and $\Sigma; \widehat{c}_n, P_n, \widehat{c}_{n+1} \vDash P_{n+1}$ hold. Since this not possible for every c and P, one can try to strengthen the induction hypothesis by, for example, using k-induction [6,28].

3 The LAMA Language

We introduce the intermediate language LAMA *(= Low Abstraction & Mode Automata)* to bridge the gap between the numerous and complex concepts offered in the SCADE language and the encoding as a set of formulas that can be delivered to an SMT-solver.

LAMA supports a reduced set of language concepts only, but structured data types and automata are included as they are promising for optimizations when transferred to the SMT framework. In this sense, LAMA extends and varies from NBAC [21]: LAMA automata allow for hierarchical and parallel composition, local dataflow may be assigned to modes, i.e. automata states, at each level. LAMA automata are inspired by mode automata [24], but with the difference that the LAMA transitions semantics corresponds to strong transitions as used in safe state machines [2] in SCADE.

A LAMA program consists of a collection of declarations of types, constants, input, local, and state variables and nodes, a (global) dataflow, initializations, assertions and an invariant. A node is declared by its name and its input and output parameters. It may contain a set of subnodes $\mathcal{N}$, its own local and state variables V, a local flow F, initializations S_0, automata definitions $\mathcal{A}$, and an invariant *Inv*. A node is denoted by $N = (\textbf{node}\ x\ y\ \mathcal{N}\ V\ F\ S_0\ \mathcal{A}\ Inv)$ in what we call abstract syntax; the concrete syntax is shown in the example in Fig. 3. An automaton $A = (\textbf{automaton}\ L_A\ l_0\ E_A) \in \mathcal{A}$ consists of a collection L_A of modes (**location** in the concrete LAMA syntax) and an initial mode l_0. The body contains the transitions (**edge**) E_A between the modes.

In case a variable is not explicitly defined in each mode, the **default** block is used to define a default assignment. The usage of a node is denoted by (**use** N $t_1 \dots t_k$) in LAMA where t_i are the actual parameter terms.

A dataflow consists of local variable definitions and the initialization and transition definition for state variables.

The definition of the next state's value of a variable[7] is denoted by „' " (see line 6, Fig. 3). In order to deal with the SCADE operators fby and pre either a **transition** definition will be used, or an automata declaration is introduced (see Sec. 4 for details). LAMA expressions may use the usual logical, arithmetical and relational operators, projections (for product types), and pattern matching for user defined enumerations. Their use, as well as the full syntax of LAMA can be found in appendix A.1 in [5].

The scope of a variable is exactly the block in which it is declared (excluding inner blocks) with the exception of globally declared enumerations and constants.

The Type System. The syntax and semantics of LAMA types are shown in Fig. 4. The LAMA typing rules follow the Cardelli's ideas [7], the details are given in appendix A.2 in [5].

Causality Analysis. As for SCADE, LAMA programs have to be causal, meaning that the definition of a variable for the current time instant must not instantaneously depend on itself. In order to check causality, a dependency graph of the variables is constructed. If this yields a strict (evaluation) order on the variables, the program is causal. Our approach for LAMA is similar to the causality check in SCADE.

[7] i.e., the next value within the stream associated with x in the LAMA semantics.

```
1  nodes
2    node UpDown () returns (xo : int) let
3      local x1 : int;
4      state x : int;
5      definition xo = x1;
6      transition x' = x1;
7
8      automaton let
9        location A let definition x1 = (+ x 1); tel
10       location B let definition x1 = (- x 1); tel
11       initial A;
12       edge (A, B) : (= x 10);
13       edge (B, A) : (= x 0);
14       edge (A, A) : (not (= x 10));
15       edge (B, B) : (not (= x 0));
16     tel
17
18     initial x = (- 1);
19   tel
20 local x : int;
21 definition x = (use UpDown);
22
23 invariant
24   (and (>= x 0) (<= x 10)); -- range: 0 to 10
```

Fig. 3. UpDown counter example adapted from [24]

3.1 Dynamic Semantics

The internal state space of a LAMA program P is denoted $S = \prod_j [\![T_j^{int}]\!]$ where T_j^{int} is the type of a state variable v_j. The space of the input and output values is denoted $X = \prod_k [\![T_k^{in}]\!]$ and $Y = \prod_l [\![T_l^{out}]\!]$. As the definition of a variable may depend on the current modes of the automata, the semantics takes the modes into account: $Q = \prod_{A \in \mathcal{A}_P} L_A$.

The semantics of P is then given by a stream transformation $[\![P]\!]\colon X^\omega \to Y^\omega$ defined by a Mealy machine (Sec. 2.4) $c_P\colon S \times Q \times X \to S \times Q \times Y$ on the state space $S \times Q$. The mapping $[\![P]\!]$ is defined by the iteration of c_P: let x be an input stream, we put $(s_{n+1}, q_{n+1}, y_n) = c_P(s_n, q_n, x_n)$. This sequence starts at $(s_0, q_0) = ([\![S_0]\!], (l_0^A)_{A \in \mathcal{A}_P})$, the semantics of the initialization predicate, and initial modes of all automata. Using this iteration, we define $[\![P]\!](x)(n) = y_n$. The LAMA semantics for the dataflow part coincides with the LUSTRE semantics.

For the sake of brevity, we discuss the automata semantics only informally: In LAMA a node N may contain a collection $\mathcal{A}$ of automata. If a flow refers to N in a **use**-construct, then N's automata are considered to run in parallel at the same hierarchical level. However, within a location l of any automaton $A \in \mathcal{A}$ a flow may use a subnode $N.M$ that again may contain a collection $M.\mathcal{B}$ of automata. The automata $B \in M.\mathcal{B}$ are the counterpart of subautomata residing within the state of a state machine in SCADE (see Sec. 4 for the translation).

$$
\begin{aligned}
\langle \textit{Type} \rangle &::= \langle \textit{BaseType} \rangle \\
&\mid \langle \textit{Identifier} \rangle \\
&\mid \langle \textit{BaseType} \rangle \texttt{\^{}}\ n \\
&\mid \texttt{(\#}\ T_1 \dots T_n\ \texttt{)} \\
\langle \textit{BaseType} \rangle &::= \texttt{bool} \\
&\mid \texttt{int} \\
&\mid \texttt{real} \\
&\mid \texttt{sint}\ [n] \\
&\mid \texttt{uint}\ [n]
\end{aligned}
\qquad
\begin{aligned}
[\![\textbf{bool}]\!]\Sigma &= \mathbb{B} \\
[\![\textbf{int}]\!]\Sigma &= \mathbb{Z} \\
[\![\textbf{real}]\!]\Sigma &= \mathbb{Q} \\
[\![\textbf{sint}[n]]\!]\Sigma &= \{-2^{n-1}, \dots, 2^{n-1}-1\} \\
[\![\textbf{uint}[n]]\!]\Sigma &= \{0, \dots, 2^n - 1\} \\
[\![\textbf{x}]\!]\Sigma &= \Sigma(x) \\
[\![T^{\wedge} n]\!]\Sigma &= [\![(\#\underbrace{T \dots T}_{n})]\!]\Sigma \\
[\![(\#T_1 \dots T_n)]\!]\Sigma &= \prod_{i=0}^{n} ([\![T_i]\!]\Sigma)
\end{aligned}
$$

Fig. 4. Syntax and semantics of LAMA types

For each automaton we distinguish between the *selected* mode at which the n-th step is assumed to start and the *active* mode that is executed at step n, $n \geq 0$. Let us assume node N is evaluated at step n. For each automaton $A \in \mathcal{A}$ the selected mode $m_{A,n}$ is considered and the most prior outgoing transition, whose guard evaluates to true, is determined. If such a transition exists, it is executed and its target m'_A is said to be the *active* mode of A in step n. Otherwise the selected mode is set active, i.e. $m'_A = m_{A,n}$. This corresponds to strong transition semantics and giving outermost transitions priority as in SCADE. Now the flow definitions are evaluated for the active mode m'_A. In case the flow of m'_A makes **use** of a subnode with automata $B \in M.\mathcal{B}$, the selected modes m_B of all B are evaluated for outgoing transitions recursively until the innermost automata are reached. The result of the flow evaluation contributes to the next step's state variables s_{n+1}. Finally, the next step's *selected* mode $m_{A,n+1}$ is set to m'_A for all automata residing in N.

Comparison with SCADE. SCADE offers a lot more language concepts most of which are translated to LAMA as explained in Sec. 4. Some concepts are not handled yet, but left for a future extension of the translation: Among these are the basic type **char**, records, and type variables, sensors, signals, clocks, and probes. Functions, which can be translated to nodes easily, static input, and the **where** ... **numeric** construct, which allows to declare polymorphic operators over numeric types, are missing. Within equations **guarantee**, **handle**, and **returns** are not handled yet, whereas in automata, dataflow cannot be assigned to transitions, **synchro**-transitions and **final** states are missing, as well as branching transitions. The sequential operators **when** and **merge** and clocked expressions are not supported. The use of higher order operators and clocked uses of operators are left out. Tuples, some array operations and structs can be easily handled.

Let us point out that even though the existing implementation cannot yet handle the full SCADE syntax, all missing language constructs can be reduced to the existing LAMA syntax in a straightforward way. Only for operator casts some primitive operators should be added to LAMA. Moreover, our translation does support a sufficiently large

fragment of SCADE that allows to perform experiments on industrial relevant models such as the ones considered in Sec. 6.

4 Translating SCADE to LAMA

We are now going to describe the translation of a SCADE model to LAMA. Due to space constraints we can only sketch the general principles and indicate where the subtleties of the translation arise; a detailed description can be found in [5]. We also must assume that the reader is sufficiently familiar with the SCADE language (see [13]). Fig. 3 shows (a simplified) form of the translation result of the SCADE model from Fig. 1.

SCADE operators are translated to LAMA nodes. Note though that each instance of a SCADE operator has its own state memory. Thus, for every instance of a SCADE operator N a copy of the translation of N with a fresh name is generated in LAMA. Within a SCADE operator there are state independent (without any synchronous state machines) and state dependent dataflows, and these must be handled separately.

State independent dataflow. Logical and arithmetic base operations as well as variables, constants, array functions and if-then-else of SCADE have counterparts in LAMA, and are hence translated directly. Stream operators are handled as follows: each fby is replaced by a chain of pre and $\rightarrow$ (init) operators and then translated. For the translation of $\rightarrow$ and pre one has to distinguish several cases. In the first case of a SCADE statement $x = M \rightarrow \mathsf{pre}\, N$ with $M = c$ a constant expression and N not containing $\rightarrow$ or pre, one translates this as a LAMA flow: **initial** x = c; **transition** x' = N; (the second case $x = \mathsf{pre}\, M$ is handled by simply omitting the initialization of x – this is correct in LAMA if the original SCADE operator was correct). In the third case where M is not constant the **initial** statement is not allowed in LAMA and so the translation yields an automaton with three modes $\mathsf{dummy} \xrightarrow{\mathsf{true}} \mathsf{init} \xrightarrow{\mathsf{true}} \mathsf{run}$ where in init we have x = M; and in state run we have x = N. Finally, the remaining cases $x = M$ are treated by unrolling, i.e., M is rewritten so that one of the first three cases can be applied (see [5]).

State dependent dataflow. SCADE synchronous state machines are translated to LAMA state machines. Hierarchy of state machines in SCADE is handled by introducing LAMA nodes for subautomata. For example, let s be a state of a SCADE state machine containing another state machine M_s with states s_1, s_2 that read variables a, b, c and write to variable x. Then state s is translated to a state containing a statement $x = (\textbf{use}\ N_s\ a\ b\ c)$, where the LAMA node N_s contains the translation of state machine M_s.

Recall that SCADE knows several types of transitions between states of synchronous state machines. The *strong* transitions have the same semantics as transitions in LAMA and are translated directly, whereas *weak* transitions need a special treatment. In particular our translation needs to carefully handle several cases where a state has both types of transitions entering and/or leaving the state (see [5] for details). SCADE also distinguishes *restart* and *resume* transitions. The former leads to the initialization of all flows in the target state; they are translated by essentially transforming them into resume transitions, which have the same semantics in LAMA and can be translated directly.

We omit the description of the translation of default declarations as well as of pre and last within states. There are also some derived language constructs in SCADE that are translated by first replacing them by equivalent SCADE constructs whose translation we already explained; this concerns: the fby operator (mentioned previously), if-blocks (replaced by state machines), when-match-blocks (replaced by a case switch and an if-block) and the times operator.

Finally, let us mention two easy optimizations that are performed in this translation step: (1) pre operators are brought to the root of expressions as much as possible (e.g. by using the *distributive law* $f(\mathsf{pre}\, M_1, \ldots, \mathsf{pre}\, M_n) \equiv \mathsf{pre}\, f(M_1, \ldots, M_n)$ that holds for every non stream operator f); (2) the elimination of auxiliary variables in textual SCADE code (especially when it is obtained from graphical models) by inlining. Both techniques reduce the number of state variables in the SMT instances obtained in the next transformation step and so lead to a smaller problem for the SMT solver at the end.

5 Translating LAMA to SMT

In this section we are going to translate a given LAMA program into a set of SMT-formulas that we can use to verify the invariant the program comes with. This is done by first translating the nodes (recursively) into Mealy machines, see Sec. 2.4, and then constructing another Mealy machine for the data flow of the program. The translation process yields one machine per variable and automaton, where a machine can use all inputs and the previous state of all machines in the current scope. More precisely, we are going to construct a signature Σ and formulas dependent on the step n, such that every symbol in Σ, except for input symbols, is defined by one formula. The symbols for each state variable and automaton have a stream type (over the type of the variable). One formula D_x then defines x at position $n+1$, possibly using other symbols at position n. This translation is easier to implement and use than constructing one machine describing all state variables/automata at once.

5.1 SMT-Formulas from Nodes

Assume that we are given a LAMA node $N = (\textbf{node}\ x\ y\ \mathcal{N}\ V\ F\ S_0\ \mathcal{A}\ P)$ of type $X \to Y$. We add symbols $x : [\![X]\!]^\omega$ and $y : [\![Y]\!]^\omega$ to the signature Σ, where one should note that X and Y may be product types if the node has several inputs or outputs. For every variable $(v : T) \in V$ of type T we add another symbol $v : [\![T]\!]^\omega$ to the signature Σ. Finally, for each $A = (\textbf{automaton}\ L_A\ l_0\ E_A) \in \mathcal{A}$ we add two more symbols $act_A, sel_A : T(L_A)^\omega$ to Σ. Here we use a type $T(L_A)$ that encodes the modes of A, for example, using integers or bitvectors.

An equation $(v = M) \in F$ (where v can be an output y) gives rise to a formula $D_v := \lambda n.\ v(n) \equiv M(n)$, a state transition $s' = M \in F$ on the other hand defines $D_s := \lambda n.\ s(n+1) \equiv M(n)$. The initial condition for s at 0 is given by the formula $I_s := (s(0) \equiv a)$ for $s = a \in S_0$. We describe the translation of LAMA terms M in Sec. 5.2.

The symbols $act_A, sel_A : T(L_A)^\omega$ for an automaton $A \in \mathcal{A}$ represent the active and the selected mode, respectively. Since we are using strong transition semantics,

sel_A will always be defined by the formula $sel_A(n+1) \equiv act_A(n)$ and the initial condition $I_A := (sel_A(0) \equiv l_0)$. The symbol act_A on the other hand is assigned the active mode of the automaton *in the current step*:

$$act_A(n) \equiv next(sel_A, E_A)(n),$$

where $next$ returns the active mode (see Sec. 3.1). Depending on the active mode we select the used computation for a variable v:

$$D_v := \lambda n.\ v(n) \equiv match(act_A, L_A, v)(n).$$

Here $match$ selects the used flow for v, depending on the active mode:

$$\begin{aligned} match(act_A, L_A, v) := \lambda n.\ & \text{if } act_A(n) \equiv l_1 \text{ then } M_1(n) \\ & \dots \\ & \text{else if } act_A(n) \equiv l_{n-1} \text{ then } M_{n-1}(n) \\ & \text{else } M_n(n) \\ \text{for } (l_i, F_{l_i}) \in L_A \text{ and equations } & (v = M_i) \in F_{l_i}. \end{aligned}$$

Finally, we tie everything together by formulas describing the flow of N using an activation condition e_N ("enable N"):

$$D_N := \lambda n.\ \text{if } e_N(n) \text{ then } \bigwedge_{v \in V} D_v(n) \text{ else } \bigwedge_{s \in state(V)} Id_s(n)$$

$$I_N := \bigwedge_{s \in state(V)} I_s$$

using the formula $Id_s(n) = (s(n+1) \equiv s(n))$ in case the dataflow of N is disabled.

The activation condition is only relevant in the case where a node is used inside a mode, since its dataflow is independently generated and shall only be active, if the mode in which the node is used is active. Let $l \in L_A$ be the mode in which a node N is used, then e_N is simply $e_N := \lambda n.\ act_A(n) \equiv l$.

Example 5.1. We translate here the node UpDown from Sec. 3. The resulting signature is $\Sigma = \{xo : \mathbb{N}^\omega, x1 : \mathbb{N}^\omega, x : \mathbb{N}^\omega, act_A, sel_A : {T_A}^\omega\}$ with $T_A = \{1, 2\}$. The state variables are defined by the formulas

$$\begin{aligned} D_{xo} &:= \lambda n.\ xo(n) \equiv x1(n) \\ D_x &:= \lambda n.\ x(n+1) \equiv x1(n) \\ D_{x1} &:= \lambda n.\ x1(n) \equiv (\text{if } act_A(n) \equiv 1 \text{ then } x(n) + 1 \text{ else } x(n) - 1) \end{aligned}$$

5.2 Translating Dataflow

There are two kinds of right-hand-sides one can have in the dataflow of LAMA: expressions or the use of a node. We will not describe the translation of LAMA expressions into SMT here, since this is just a point-wise application on streams. This leaves us with

the case of (**use** N t) for a node identifier N and the argument t. Recall that we added symbols for input and output, say x, y, to the signature Σ of node N. The use of a node is driven by "connecting" x to t:

$$D_x := \lambda n.\ x(n) \equiv t(n).$$

The term (**use** N t) is translated to the symbol y, i.e., an equation $v = (\textbf{use}\ N\ t)$ is translated to $D_v := v(n) \equiv y(n)$ in Sec. 5.1.

5.3 SMT-Formulas from Programs

Finally, we translate the top level of a LAMA program. Such a program consists of node declarations and dataflow, which are handled as described in the previous subsections, and an invariant P. This P is immediately translated to an SMT-formula and hence we can check its validity according to Sec. 2.4, using the formulas D_N and D_v from the above translation in lieu of Iter from (1).

5.4 Correctness of the Translations

We briefly mention here a possible strategy for proving the correctness of the given translations. A scheme of the translation steps we have given is shown in the top row

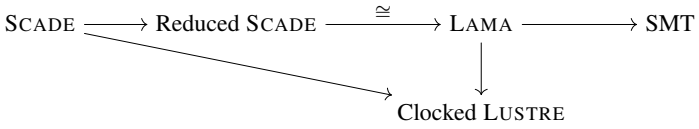

Fig. 5. Translation steps

of Fig. 5. In [10] Colaço et al. gave semantics to a fragment of SCADE, including safe state machines, by translating it into a variant of LUSTRE with clocks. Thus, a possible strategy would be to translate LAMA into this language as well and show that our translation yields equally behaving programs. If, moreover, we give semantics to LUSTRE with clocks in terms of Mealy machines, we can also prove our translation into SMT formulas correct.

However, this proof has not been carried out so far, and we leave it as future work. Instead, we have taken a more practical approach for the time being, in that we have compared our implementation to the SCADE Design Verifier, see Sec. 6.

6 Implementation and Experiments

In this section we describe a first implementation of the transformation of SCADE programs to SMT instances. We also present results of first experiments using our implementation on an industrial SCADE model. For this we reproduce verification results that were already obtained in [20] using SCADE DV, and we compare the running times.

Our implementation of the LAMA framework consists of the following components, all of which are written in the functional programming language *Haskell*:

- A library to parse, manipulate and render LAMA-programs: *language-lama*.
- A parser library for the SCADE (textual) language: *language-scade*.[8]
- An SMT interface abstraction which allows us to seamlessly use multiple different SMT solvers called *smtlib2*.[9]
- A program to translate SCADE- into LAMA-programs: *scade2lama*.
- An interpreter for the LAMA language, which can be used to interactively run LAMA-programs: *lama-interpreter*.
- The verification component which verifies LAMA-programs by translating them into SMT: *lamasmt*.

All components are available under liberal free-software licenses.[10] The general workflow and the interaction of the components can be seen in Fig. 6.

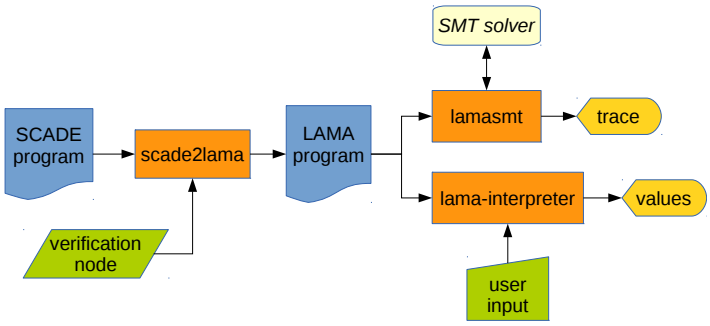

Fig. 6. Framework components and their interaction

SCADE programs are translated into the LAMA language using *scade2lama*. The user has to supply the name of the SCADE node whose properties shall be verified. The resulting LAMA program can then be formally verified using the *lamasmt* tool, which uses either bounded model checking or k-induction. It communicates with the SMT solver via the *smtlib2* library and produces a counterexample trace or states that the property holds for the node (this is only possible when using k-induction). If the k-induction is not able to prove the property within a specifiable depth, *lamasmt* can produce candidate-counterexamples from the induction step. These may be used later to generate lemmas, to strengthen the induction hypothesis, e.g. by adapting ideas from Champion et al. [9].

[8] Available from `https://github.com/hguenther/language-scade`.
[9] Available from `https://github.com/hguenther/smtlib2`.
[10] Available from `https://github.com/hbasold/lama`

6.1 Benchmarks

We evaluated the performance of the *lamasmt* verification tool by applying it to a model of a level crossing system.[11] The details and the descriptions of hazards and the fault modes of this model can be found in [20].

We compared our tool with the SCADE DV, the proprietary verification tool bundled with SCADE. Like our tool, it can be used to verify the correctness of properties (called "proof strategy") or as a bounded model checker (called "debug strategy"). First, we compared SCADE DV's proof strategy against *lamasmt* with k-induction. We use the

	SCADE DV		LAMA	
env. model	proven	time	proven	time
(1)	yes	12s	no (depth 27)	5h
(2)	no (depth 42)	205m	no (depth 46)	27h
(3)	yes	23h	no (depth 50)	68h

Table 1. Comparing proof strategies

three environment models described in the study [20]: in model (1) a train is constantly occupying one of the two available tracks, while in model (2) a train can appear and disappear on track one at random, and in (3) a single train passes through track one for 40 cycles. The property to prove for all these environment models is that no train runs through an unprotected level crossing.

As Table 1 shows, the pure k-induction strategy does not work very well on the provided model. This confirms an observation made in [9] that without further heuristics k-induction does not scale up very well as a property may require a k that leads to a too large unfolding of the model, or it may not be k-inductive at all.

We also compared the BMC strategies of SCADE DV and LAMA on the five described fault modes of the model. Each fault mode was treated using environment model (1). The fault modes describe the following behaviours: a defect of a traffic light (*L1* and *L3*), a mis- or false-detection by a barrier sensor (*BS13* and *BS11* resp.), and a barrier that got stuck or is misbehaving (*B7* and *B9* resp.). We can see in Table 2 that both tools are able to find the three first fault modes and unable to find counter-examples for the last two. The found faults occurred at depth 27 in all cases.

While the performance of *lamasmt* does not yet match that of the SCADE DV, the results nonetheless give us an indication that the implementation is indeed correct: There are no false errors being found, nor are any hazards undetected by our implementation. In the benchmarks, time and memory used by the intermediate translation steps are negligible, most of the time is taken by the SMT-solver.

[11] All benchmarks were performed on an Intel® Core™ Duo CPU P9600 @ 2.53 GHz with 4 GB of RAM.

	SCADE DV		LAMA	
bug	found	time	found	time
L1	yes	1s	yes	422s
BS13	yes	1s	yes	491s
B7 + BS11	yes	11s	yes	418s
B9	no (depth 35)	10s	no (depth 35)	24m
L3 + BS11	no (depth 50)	62s	no (depth 30)	13m

Table 2. Comparing BMC strategies

7 Conclusion and Future Work

In this work, we developed an open experimentation platform for the verification of SCADE programs based on recent SMT technology. The verification process has been divided into several translation steps. First, SCADE programs are transformed into a small subset of SCADE that corresponds to programs in the intermediate language LAMA. LAMA keeps a few abstractions of SCADE, which are promising to facilitate optimization of the actual verification. After translating SCADE- into LAMA-programs, the resulting LAMA programs can almost directly be interpreted as sets of SMT formulas, describing transition steps. This step-wise description can then be used to find counterexample using bounded model checking or to prove predicates using k-induction.

These translation steps have been implemented as open source software. This software can already find the same counterexamples in a medium-sized design developed in industry as the proprietary SCADE Design Verifier, which comes with the SCADE suite. However, the verification procedure and performance are still lacking behind.

Since the developed software is meant to be an experimentation platform, future work obviously includes optimizing better verification techniques and the translations. This might effect the intermediate language LAMA itself.

References

1. Abdulla, P.A., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing safe, reliable systems using scade. In: LNCS 4313. pp. 115–129. ISoLA'04, Springer-Verlag (2006)
2. André, C.: Semantics of S.S.M (Safe State Machine). Tech. Rep. UMR 6070, I3S Laboratory, University of Nice-Sophia Antipolis (2003), available at `http://rw4.cs.uni-saarland.de/teaching/esd07/papers/SSMsemantics.pdf`
3. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, chap. 26, pp. 825–885. IOS Press (2009)
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Proc. 8th Intern.Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
5. Basold, H.: Transformationen von Scade-Modellen zur SMT-basierten Verifikation. Master's thesis, TU Braunschweig (2012), `http://arxiv.org/abs/1403.2752`
6. Bjesse, P., Claessen, K.: SAT-based verification without state space traversal. In: Jr., W.A.H., Johnson, S.D. (eds.) Proc. FMCAD. LNCS, vol. 1954, pp. 409–426 (2000)

7. Cardelli, L.: Type systems. In: Tucker, A.B. (ed.) CRC Handbook of Computer Science and Engineering, chap. 97. Chapman and Hall (2004)
8. CENELEC: EN 50128 – Railway Applications – Software for Railway Control and Protection Systems. European Standard. (2012)
9. Champion, A., Delmas, R., Dierkes, M.: Generating property-directed potential invariants by backward analysis. In: Proc. FTSCS. EPTCS, vol. 105, pp. 22–38 (2012)
10. Colaço, J.l., Pagano, B., Pouzet, M.: A conservative extension of synchronous data-flow with state machines. In: EMSOFT. pp. 173–182. ACM Press (2005)
11. DO-178B: Software considerations in airborne systems and equipment certification (Dec 2011)
12. Dutertre, B., de Moura, L.: The YICES SMT solver. Tech. rep., SRI Int. (2006)
13. Esterel: Scade language reference manual (2011)
14. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. Electr. Notes Theor. Comput. Sci. 144(1), 19–33 (2006)
15. Hagen, G.: Verifying safety properties of Lustre programs: an SMT-based approach. Ph.D. thesis, Department of Computer Science. The University of Iowa (2008)
16. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Proc. FMCAD. pp. 1–9 (2008)
17. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proceedings of the IEEE. vol. 79:9, pp. 1305–1320. IEEE (1991)
18. Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: Proc. of AMAST'93. pp. 83–96. Workshops in Computing, Springer-Verlag, London, UK (1994)
19. Halbwachs, N., Raymond, P.: A turotial of Lustre (2002), `http://www-verimag.imag.fr/~halbwach/lustre-tutorial.html`, last accessed: Mar. 13, 2014
20. Huhn, M., Milius, S.: Observations on formal safety analysis in practice. Science of Computer Programming 80, Part A(0), 150–168 (2014)
21. Jeannet, B.: The NBAC verification/slicing tool, `http://pop-art.inrialpes.fr/people/bjeannet/nbac/index.html`, last Accessed: Feb. 17, 2014
22. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 55–62 (2011)
23. Ljung, M.: Formal modelling and automatic verification of Lustre programs using NP-Tools. Master's thesis, Prover Technology AB and Department of Teleinformatics, KTH, Stockholm (1999)
24. Maraninchi, F., Rémond, Y.: Mode-automata: About modes and states for reactive systems. In: European Symposium On Programming. Springer verlag, Lisbon (Portugal) (Mar 1998)
25. de Moura, L., Owre, S., Shankar, N.: The SAL language manual. Tech. rep., SRI International (2003), `http://sal.csl.sri.com/doc/language-report.pdf`, last accessed: Mar. 12, 2014
26. Moura, L.D., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proc. TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008)
27. Pace, G., Halbwachs, N., Raymond, P.: Counter-example generation in symbolic abstract model-checking. Int. J. Software Tools and Technology Transfer 5(2), 158–164 (2004)
28. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD. LNCS, vol. 1954, pp. 127–144. Springer (2000)
29. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of formal analysis into a model-based software development process. In: Leue, S., Merino, P. (eds.) Proc. FMICS'07. LNCS, vol. 4916, pp. 68–84. Springer (2008)

A Appendix

A.1 LAMA Type Checking

For type checking we add the meta-type **ok** that indicates that a declaration is correctly typed. Moreover, we add function types and polymorphic types to ease the handling of functions and polymorphic operators like "+" (see Fig. 7).

$$
\begin{aligned}
&\langle \textit{IntermediateType0} \rangle ::= \langle \textit{Type} \rangle \mid \texttt{ok} \mid \texttt{x} \\
&\qquad\qquad \mid \quad \langle \textit{IntermediateType0} \rangle \Rightarrow \langle \textit{IntermediateType0} \rangle \\
&\langle \textit{IntermediateType} \rangle ::= \langle \textit{IntermediateType0} \rangle \mid \forall\ \texttt{x}\ \texttt{:}\ \langle \textit{Universe} \rangle\ \texttt{.}\ \langle \textit{IntermediateType} \rangle \\
&\langle \textit{Universe} \rangle ::= \texttt{Type} \mid \texttt{Num}
\end{aligned}
$$

Fig. 7. Intermediate types used in type checking

$\langle \textit{Intermediate0} \rangle$ denotes the quantifier-free level. By restricting the introduction of quantifiers to $\langle \textit{Intermediate} \rangle$ the construction of types with polymorphic arguments is prevented: Any type variable x is assigned to a concrete type, that is either a numeric type $Num \ni$ **bool**, **int**, **real**, ... or a user-defined type $Type \supset Num$.

Σ denotes the *type environment* which assigns a type variable to its type, Δ denotes the *constant environment* which assigns each constant its value, and Γ the *variable environment*, respectively. The typing rules follow the ideas given in [7]: We use $\Sigma \vdash D \therefore \Delta$ to denote that the declaration D is well-typed and yields signature Δ. For the body of a LAMA program we use $\Sigma, \Delta, \Gamma \vdash E : \mathbf{T}$ to denote that expression E is well-typed with type **T**. A LAMA program consists of declarations of types (T), constants (C), and inputs (In), declarations of local and state variables and nodes summarized as (D), a dataflow (F), an initialization (S_0), an assertion (A) and an invariant (Inv). A program is well-typed if all of its parts are well-typed, see Fig. 8. Most of the typing rules are straightforward, as e.g. the rule for type declarations.

$$
\frac{\begin{array}{cc} \vdash T \therefore \Sigma & \Sigma \vdash C \therefore \Delta \\ \Sigma \vdash In \therefore \Gamma_1 & \Sigma \vdash D \therefore \Gamma_2 \\ \Gamma = \Gamma_1 \cup \Gamma_2 & \\ \Sigma, \Delta, \Gamma \vdash F : \mathbf{ok} & \Sigma, \Delta, \Gamma \vdash S_0 : \mathbf{ok} \\ \Sigma, \Delta, \Gamma \vdash A : \mathbf{ok} & \Sigma, \Delta, \Gamma \vdash Inv : \mathbf{ok} \end{array}}{\vdash T\ C\ In\ D\ F\ S_0\ A\ Inv : \mathbf{ok}}\ \text{(program)}
$$

$$
\frac{\vdash T_1 \therefore \Sigma_1, \ldots, T_n \therefore \Sigma_n}{\vdash \texttt{typedef}\ \ T_1 \ldots T_n\ \ \texttt{;} \therefore \bigcup_{i=1}^{n} \Sigma_i}\ \text{(typedef)}
$$

Fig. 8. Typing rule for programs and type declarations

We just mention the typing rule for a **node**. Again all parts have to be well-typed in order to deduce that a node is well-typed. Notably, a node is assigned a function

type that maps the inputs to the outputs. The possible side effects on state variables and automata modes are not reflected in the node type. The complete type system can be found in [5].

$$\frac{\begin{array}{cc} \mathbf{T}_1 = \Sigma(t_1),\ \ldots,\ \mathbf{T}_n = \Sigma(t_n) & \Gamma_1 = \{(x_1, \mathbf{T}_1), \ldots, (x_n, \mathbf{T}_n)\} \\ \mathbf{T}'_1 = \Sigma(t'_1),\ \ldots,\ \mathbf{T}'_m = \Sigma(t'_m) & \Gamma_2 = \{(y_1, \mathbf{T}'_1), \ldots, (y_m, \mathbf{T}'_m)\} \\ \multicolumn{2}{c}{\Sigma \vdash D \therefore \Gamma_3} \\ \multicolumn{2}{c}{\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3} \\ \Sigma, \Delta, \Gamma \vdash F : \textbf{ok} & \Sigma, \Delta, \Gamma \vdash C : \textbf{ok} \\ \Sigma, \Delta, \Gamma \vdash S_0 : \textbf{ok} & \Sigma, \Delta, \Gamma \vdash A : \textbf{ok} \end{array}}{\begin{array}{c} \Sigma, \Delta \vdash \texttt{node x (}\ x_1\texttt{:}t_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{:}t_n\ \texttt{)returns (}\ y_1\texttt{:}t'_1\texttt{,}\ \ldots\texttt{,}\ y_m\texttt{:}t'_m\ \texttt{)} \\ \texttt{let}\ D\ F\ C\ \ S_0\ A\ \texttt{tel}\ :\ \textbf{(\#}\ \mathbf{T}_1, \ldots, \mathbf{T}_n\textbf{)} \Rightarrow \textbf{(\#}\ \mathbf{T}'_1, \ldots, \mathbf{T}'_m\textbf{)} \end{array}}\ \text{(node)}$$

Fig. 9. Typing rules for programs and nodes