2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops

An Invariant Preserving Transformation for PLC Models

Jan Olaf Blech, Anton Hattendorf, Jia Huang
fortiss GmbH
Guerickestraße 25, 80805 München, Germany
{blech,hattendorf,huang}@fortiss.org

***Abstract*—Many applications in the industrial control domain are safety-critical. A large number of analysis techniques to guarantee safety may be applied at different levels in the development process of a Programmable Logic Controller. The development process is typically associated with a tool chain comprising model transformations. The preservation of safety properties in model transformations is necessary to achieve a safe system. Preservation can be guaranteed by showing that invariants are preserved by transformations. Adequate transformation rules and invariant specification mechanisms are needed for this.**

We report on a transformation from Sequential Function Charts and Function Block Diagrams of the IEC 61131–3 standard to BIP. Our presentation features a description of formal syntax and semantics of the involved languages. We present transformation rules for generating BIP code out of IEC 61131–3 specifications. Based on this, we establish a notion of invariant preservation between the two languages.

I. Introduction

In the industrial control domain, many applications are mission-critical or safety-critical. For example, a misbehaving robotic arm might lead to large damage to the plant or even to the operating human. Thus, safety analysis of such systems is highly desirable. The use of formal specification and modeling techniques for analysis, transformation, and verification of the involved software is a prerequisite for the application of analysis and verification tools and the use of their results.

The IEC 61131–3 standard [21] describes a set of languages widely used in the industrial control domain to write programs for Programmable Logic Controllers (PLC). The standard defines in total five languages. In this paper we consider a subset of IEC 61131–3 languages supported by EasyLab [2], which is a model-based development tool that features a graphical interface and several simulation/debugging facilities to allow efficient design of control programs. EasyLab supports a subset of the Sequential Function Chart (SFC) language and the Function Block Diagram (FBD) languages. We provide a formal definition of this subset of IEC 61131-3 and present a transformation specification to the BIP [4] language. BIP is a language to describe component based systems. It is based on state transition systems that represent components. Components can be connected with each other. Via connectors they can interact and synchronize. Our transformation to BIP is designed to preserve the behavior of the PLC while using as much of BIPs parallelism as possible. We present a description formalism for invariants on the IEC 61131–3 language and show how invariants can be transformed between the two languages.

A. Our Tool Chain

We are addressing a development tool chain (Figure 1) where models get transformed into other models and finally into deployable machine code. The first stage comprises the modeling of a system using EasyLab and the IEC 61131–3 standard. There are two possible ways to get deployable machine code out of an IEC 61131–3 specification:

1) We can transform the model into the BIP language. An invariant and deadlock discovery tool (D-Finder [5]) can be invoked on the BIP level and then we can generate code from the BIP model (BIP code generation).
2) On the other hand, we can directly generate code from the IEC 61131–3 model (EasyLab code generation).

While the way via BIP is more suitable during the development phase, for verification and simulation purposes, generating directly code from the IEC 61131–3 language can be used for a final product since it might be more efficient. It is possible to specify safety properties either on the IEC 61131–3 level or on the BIP language. The languages from the IEC 61131–3 standard and BIP where designed for different purposes. The IEC 61131–3 languages are classical examples of domain specific languages, while BIP is a general modeling language. Comparing these languages with respect to abstraction possibilities and expressiveness goes beyond the scope of this paper.

In addition to the transformation we are interested in lifting invariants representing safety properties between IEC 61131–3 and BIP. In particular we are interested in two different use-cases.

Use-Case 1: Lifting Invariant Properties from BIP to IEC 61131–3: Suppose we want to use the EasyLab code generation but still want to use invariants and safety properties discovered on a corresponding BIP representation. In this case we have to perform a lifting of invariants and connected safety properties that might have been discovered by D-Finder on the BIP representation back to the original IEC 61131–3 model. This use-case aims at guaranteeing that discovered safety properties do also hold if we use the EasyLab code generation. In order to perform a lifting of invariants that ensures safety properties we have to ensure the following conditions:

978-0-7695-4377-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ISORCW.2011.46

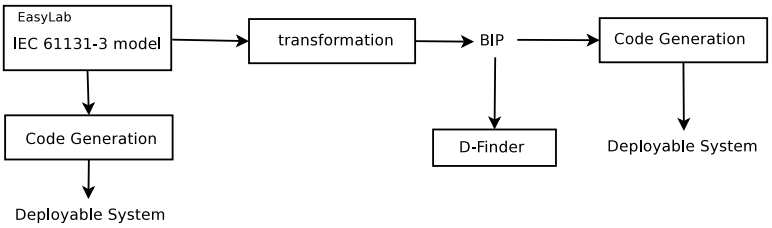

Fig. 1: Our Tool Chain

- The IEC 61131–3 model allows at least as much behavior as the corresponding BIP model.
- Invariants on the BIP model are at most as strong as corresponding invariants on the IEC 61131–3 model.

The fact that invariants may only get stronger when transforming them to an IEC 61131–3 representation ensures the preservation of safety properties.

This use case would benefit from a code generation that provides some correctness guarantee. This remains future work.

Use-Case 2: Lifting Invariant Properties from IEC 61131–3 to BIP: In the second use-case we start with an invariant (e.g., a property stating that a certain state is never reached) on the IEC 61131–3 model. The following items guarantee preservation of safety properties:

- The IEC 61131–3 model allows at most as much behavior as the corresponding BIP model.
- The invariant on the IEC 61131–3 representation is at least as strong as the corresponding invariant on the BIP model.

In this work we are presenting a transformation that preserves the exact behavior of IEC 61131–3, i.e., we do not eliminate or introduce non-determinism during the transformation from IEC 61131–3 to BIP. One intention is to use it together with the tool chain sketched in [8].

B. Related Approaches

Our work is influenced by approaches to guarantee the correctness or distinct properties of compiler runs (e.g., [20], [16], [18]). Our transformation rules for the IEC 61131–3 to BIP transformation can be regarded as the definition for a compiler. Guaranteeing correctness of our transformation by using translation validation like techniques and establishing an approach to guarantee properties of the transformation similar to [6], [7] is a long term goal of our work.

The aspect of property preservation during transformations is influenced by [17]: The work presents a proof stating conditions for abstractions that preserve temporal logics properties. Our transformations may be regarded as abstractions. The invariant based safety properties regarded in this paper are a simple (but yet powerful) case of these properties.

SFCs, their semantics and the verification of properties by using model checkers have been studied in [3], [11]. This semantics formalism for SFCs was the starting point for our semantics of the IEC 61131–3 subset.

Various other BIP transformations exist that are relevant for our work. Most notably synchronous BIP [12] is a language subset of BIP that is especially suitable for the transformation of synchronous languages into BIP. The languages of IEC 61131–3 are also synchronous. Furthermore the translation from AADL to BIP has been studied [13]. Coq certificates guaranteeing correct invariants for BIP have been studied in [9].

The transformation of contract specifications and their proofs has been studied for object oriented programs [19]. In addition to these approaches, formal specification and correctness of model to model transformations have been extensively studied in the context of graph-transformations [14].

C. Overview

We define a formal semantics of the used IEC 61131–3 subset in Section II. A summary of the BIP language and its semantics is given in Section III. The transformation from IEC 61131–3 into BIP is described in Section IV. Invariants and their preservation are discussed in Section V. Finally, Section VI features a short discussion of our implementation and Section VII features a conclusion.

II. Semantics of IEC 61131–3

In this section we present the semantics of SFCs and FBDs. Both are graphical representations. The IEC 61131–3 standard leaves some semantical aspects open to concrete implementations. Unlike a programming language standard behavior of PLCs is influenced by concrete hardware elements and their physical wiring. In the following we describe a semantics definition consistent with EasyLab and the PLCs targeted for use with EasyLab. While SFCs are used to describe the overall control flow, FBDs are used to describe function-like computations and can be referenced within an SFC to perform such a computation.

A. Semantics of SFCs

The following description of SFCs builds upon [3]. SFCs comprise control locations of the system (called *steps*) and the transition of control between them. The passing of control can be restricted via *guards*. Behaviors of the program is described in *action* blocks which can be associated with steps. These actions may be realized as FBDs but can be written in any other languages described in IEC 61131-3. In the following, we first define the basic components of SFCs and then describe the composition of them.

Variables in the SFC Language: SFCs have variables that are visible to all their components, such as steps, guards and actions. We use $X = \{x_1, x_2, ..., x_n\}$ to denote the set of variables. The current values of X are described using a variable valuation function (usually denoted f in the context of this paper) of type $X \rightarrow val_X$, which assigns a value compatible to the respective data type (val_X) to each variable in X. We use $\mathcal{F}$ to denote the set of all valuation functions of X.

Action Blocks and Steps: Action blocks and steps are the basic SFC units for describing the behavior. In the case of EasyLab an action is an update function of type $\mathcal{F} \rightarrow \mathcal{F}$ which might be modeled as an FBD.

Definition 1 (Step): For a given set of actions A, a step of an SFC is a pair $s = (n, \Omega)$, where n is a unique identifier for the step and $\Omega = 2^A$ is a set of actions belonging to the step. We use $s.\Omega$ to refer to the set of actions associated with step s.

The steps can be in *inactive* or *active* state. The set of actions will be activated for execution if the associated step is active.

Guards and Transitions, SFC Definition: Steps in SFCs are connected via transitions. A transition features a guard expression.

Definition 2 (Guard): A guard g is a predicate over a valuation function. It has the type $g : \mathcal{F} \rightarrow bool$, where $\mathcal{F}$ is the set of all valuation functions. It evaluates to true if the current values of X satisfy g.

Definition 3 (Transition): A transition (t_{src}, t_g, t_{tgt}) describes the moving of control from source steps t_{src} to target steps t_{tgt}. A transition is enabled if the guard t_g evaluates to true. A transition is taken if no conflicting transition with higher priority is enabled.

Definition 4 (SFC): An SFC is a 5-tuple $\mathcal{S} = (X, A, S, S_0, T)$, where

- *X is a finite set of variables, A a finite set of actions,*
- *S is a finite set of steps, comprising action blocks which depend on A, S_0 is the set of initial steps,*
- *$T \subseteq (2^S \backslash \{\emptyset\}) \times G \times (2^S \backslash \{\emptyset\})$ is the set of transitions, where G is the set of guards,*

For some applications priorities of transitions between steps are required. In our framework we propose a realization by using appropriate guard expressions.

Example: Figure 2 depicts an example SFC model consisting of four steps and three actions. Its formal definition is:

$$\begin{aligned}\mathcal{S} = \{&\{x, y\}, \{S1, S2, S3, S4\}, \{a1, a2, a3\}, \{S1\},\\ &\{(\{S1\}, x1 < 10, \{S2\}), (\{S1\}, x > 10, \{S3\}),\\ &(\{S2\}, x > 15, \{S4\}), (\{S3\}, x < 5, \{S4\})\\ &(\{S4\}, true, \{S1\})\}\}\end{aligned}$$

1) Operational Semantics: As proposed in [3], the operational semantics for SFCs is based on configurations describing the system state :

Definition 5 (Configuration): A configuration of an SFC is a 3-tuple $c = (f, activeS, activeA)$, where f is the function describing the current values of variables, $activeS$ is the set of active steps and $activeA$ is the set of active actions.

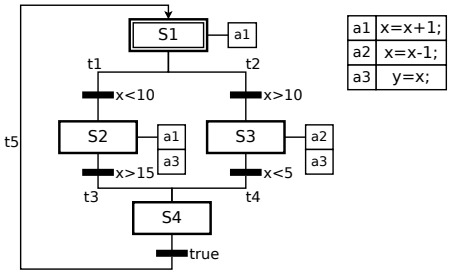

Fig. 2: An example SFC

Execution of one SFC-cycle consists of two major phases:

- Phase 1: execute actions contained in set $activeA$ and update the values of variables correspondingly;
- Phase 2: perform step transitions and update the sets of active steps $activeS$ and active actions $activeA$ for the next cycle.

In each of the two phases, the configuration of the SFC system is updated. The semantics of an SFC can then be regarded as a transition system of the configurations.

Definition 6 (Transition system of an SFC): An SFC $\mathcal{S} = (X, A, S, S_0, T)$ is associated with a transition system $\mathcal{E}(S) = (\mathcal{C}, c_0, \rightarrow)$, where $\mathcal{C}$ is the set of configurations, c_0 is the initial configuration and $\rightarrow \subseteq C \times C$ is the transition relation.

Let $[\![SM]\!]_{SFC}$ denote the set of all possible configuration transitions of an SFC SM. It can be formally defined as follows:

$$(c, c') \in [\![SM]\!]_{SFC} \;\; iff$$

$$executeAction(c, c') \vee stepTransition(c, c')$$

The term $executeAction$ and $stepTransition$ represent the two possible types of configuration transitions. Formally, for $c = (f, activeS, activeA)$, $c' = (f', activeS', activeA')$:

$$\begin{aligned}&executeAction(c, c') = \exists\ \hat{a} \in activeA\ .\ f' = \hat{a}(f)\\ &\quad \wedge activeS = activeS' \wedge activeA' = activeA \backslash \{\hat{a}\}\end{aligned}$$

$$\begin{aligned}&stepTransition(c, c') = \exists\ t \in T\ .\\ &\quad t_{src} \subseteq activeS \wedge t_g(f) \wedge f = f'\\ &\quad \wedge (activeS' = activeS \setminus t_{src} \cup t_{tgt})\\ &\quad \wedge \forall s\ \in t_{src}\ .\ activeA \cap s.\Omega = \emptyset\\ &\quad \wedge activeA' = \{a | \exists s\ .\ s \in t_{tgt} \wedge a \in s.\Omega\} \cup activeA\end{aligned}$$

The first type of transition corresponds to the execution of an active action. In a transition of the second type, the step transition guards are evaluated and the new active steps and actions are computed. The conditions on $activeA$ and $activeA'$ enforce that all actions of a step have been executed before passing control to a succeeding step. The reachable configurations of SM can then be inductively defined as follows, demanding that the initial state is reachable and each reachable configuration must be able to be reached from the initial state via valid transitions.

$ReachableConfig_{SFC}(c) =$

$$\left.\begin{array}{ll} & c = c_0 \\ \vee & \exists\ c'.ReachableConfig_{SFC}(c') \\ & \wedge\ (c', c) \in [\![SM]\!]_{SFC} \end{array}\right\} smallest\ fixpoint$$

B. Semantics of FBDs

Actions may be regarded as functions that reading some values from input variables write some values to output variables. FBDs allow the modeling of such functions in a PLC way. An FBD consists of nodes (e.g., blocks that perform a certain computation) that are connected with each other via ports and connectors.

Definition 7 (FBD): An FBD is a tuple (N, P, C, R, W) *comprising:*
- a set of nodes N *with associated ports* P*,*
- a set of directed connections $C \subseteq P \times P$*,*
- a set of variable readers $R \subseteq N$*,*
- a set of variable writers $W \subseteq N$*.*
The semantics of an FBD is defined via a fixed-point iterations where:

- Initial variables are read from the SFC configuration via the R nodes.
- Nodes where all incoming connections have values compute their value.

Circular dependencies are allowed, however, a well-formedness condition requires that a fixed-point can be computed. Once this is done, the values of W nodes are written to the succeeding configuration of the SFC.

III. BIP and its Semantics

In this section we present a subset of the BIP language. BIP (Behavior, Interaction, Priority) is a software framework designed for building embedded systems consisting of asynchronously interacting components, each specified as a non-deterministic state transition system. We discuss its semantics and an example BIP model. Parts of this section follow the presentation given in [9] building upon [4]. Tools developed for BIP comprise static analyzers and code generation.

Atomic Components: BIP models are composed of atomic components [4], [5]. An atomic component (L, P, T, V, D) is a state transition system consisting of a set of locations L, a set of ports P, a set of transitions T, and a set of variables V which are mapped to values of type D. An atomic component has a distinct state of type $L \times (V \to D)$ comprising a location and a variable valuation. The latter is a mapping from variables to their values. Transitions are of type $T \subset L \times ((V \to D) \to bool) \times ((V \to D) \to (V \to D)) \times P \times L$ They comprise a source location, a guard function, an update function, a port, and a target location. Our semantics requires that a transition from one location to another can be performed iff the guard function evaluates to true using the variable valuation in the current state. During a transition the variable valuation is updated for the succeeding state. Furthermore, it is possible to restrict transitions by putting constraints on the port involved in the interaction. Values may be exchanged during such an interaction.

Each port $p \in P$ can have an associated variable. This variable is used to exchange data between different atomic components in composed components (see below). For composition the definition of an atomic component may be augmented with such a function. The functions $\mathcal{V}(p) : P \to V \cup \{\epsilon\}$ defines the association between port and variable. If the result of $\mathcal{V}$ is ϵ, the port does not exchange data.

Composed Components: Atomic components may be glued together to form composed components. The behavior of the resulting system can be restricted by linking components with connectors. These put constraints on ports in the different atomic components. A composed component is a tuple (A, C) comprising a set of atomic components A and a set of connectors C: $C \subset (A \times P) \times 2^{A \times P}$. Connectors have the following form: $((a_s, p_s), \{(a_1, p_1), (a_2, p_2), ..., (a_n, P_n)\})$ comprising atomic components $a_x \in A; x \in \{s, 1...n\}$ and relevant ports $p_x \in P_{a_x}$ with P_{a_x} being the set of ports associated with a_x. If the connector is used to exchange data, the data is copied from a_s to all $a_i; i \in 1...n$. For connectors without data exchange there is no difference in the treatment of a_s and a_i.

In an extended version the connectors may contain guard functions, depending on the variable valuations of the linked components, and update functions which are performed on these variable valuations and mechanisms for value exchange between components.

Gluing components together by using connectors realizes weak and strong synchronizations as well as broadcasts between components. Update functions on the involved variable valuations are used to pass values between components.

Semantics of Composed Components: The state of a composed component is the product of its atomic components' states: $(L_1 \times (V_1 \to D_1)) \times \ldots \times (L_m \times (V_m \to D_m))$. A state transition relation $[\![BM]\!]_{BIP}$ is defined upon them for a composed component $BM = (A, C)$. We assume an indexing of atomic components $A = \{a_1, ..., a_m\}$

$$(((l_1, x_1), ..., (l_m, x_m)) \ , \ ((l'_1, x'_1), ..., (l'_m, x'_m))) \in [\![BM]\!]_{BIP}$$

iff

$$\begin{aligned}
&\exists (c_s, C_r) \in C \ . \ \forall i \in \{1...m\} \ . \\
&\big(\exists j \in \{1...m\} \ . \ (\exists p : (a_j, p) = c_s \wedge \mathcal{V}(p) = v_j) \\
&\wedge (l_i, g_i, f_i, p_i, l'_i) \in a_i \ . \ T \wedge g_i(x_i) \\
&\wedge \mathcal{V}(p_i) = v_i \wedge x'_i = f_i(x_i)[v_i \leftarrow x_j(v_j)] \\
&\wedge (a_i, p_i) \in (\{c_s\} \cup C_r)\big) \\
\vee &\big(l_i = l'_i \wedge \neg(\exists p \ . \ (a_i, p) \in (\{c_s\} \cup C_r)) \wedge x_i = x'_i\big)
\end{aligned}$$

$a_i.T$ denotes the set of transitions associated with component a_i.

Reachable States of BIP models: The set of reachable states for a BIP model BM for a given initial state s_0 is defined by a predicate R_{BM} via the following inductive rules:

$$\frac{}{R_{BM}(s_0)} \qquad \frac{R_{BM}(s) \quad (s, s') \in [\![BM]\!]_{BIP}}{R_{BM}(s')}$$

The first rule says that the initial state is reachable. The second inference rule captures the transition behavior of BIP using the transition relation.

An Example

Figure 3 shows a temperature control system modeled in BIP, which has been well discussed in the literature (e.g., [5], [1], [15]).

The system controls the cooling of a reactor by moving two independent control rods. Each one has its own timer t_1, t_2. After the usage of a rod there is a timeout t_{max} until it can be reused again. The goal is to keep the temperature θ between θ_{min} and θ_{max}. When the temperature reaches the maximum

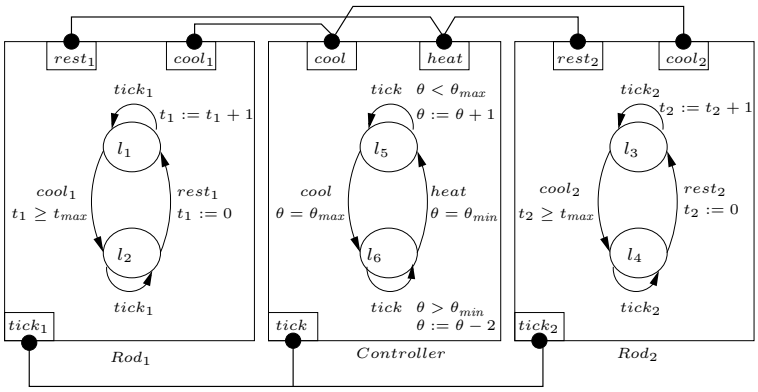

Fig. 3: Temperature Control System

value, one of the rods has to be used for cooling. The BIP model comprises three atomic components: one for each rod and one for the controller. Each contains a state transition system. Transitions can be labeled with guard conditions, valuation function updates, and a port. The components interact via ports thereby realizing cooling, heating, and time elapsing interactions. In the figure, possible interactions are indicated by arcs connecting ports from different components. These require that either in every connected component a transition labeled with the connected port must be taken in parallel or none of these transitions is taken. For example, the time elapsing $tick$ transitions must be taken in each component in the same parallel step ($tick1$, $tick$, and $tick2$). Depending on the values of θ_{max}, θ_{min}, and t_{max} the system might either contain a deadlock or not.

Here we present an example invariant for the example model with $\theta_{max} = 1000$, $\theta_{min} = 100$, and $t_{max} = 3600$:

$$(at_{l_5} \wedge 100 \leq \theta \leq 1000) \vee (at_{l_6} \wedge 100 \leq \theta \leq 1000)$$

This invariant states that the temperature will always be between 100 and 1000. at_i is a predicate denoting the fact that we are at location i in a component.

IV. IEC 61131–3 to BIP Transformation

In this section we describe the transformation from IEC 61131–3 models into BIP. We concentrate on the transformation of SFCs into BIP. Since we are interested in the preservation of structural invariants, an explicit formal transformation specification of structural entities is required. This paper presents a brief transformation specification, a more detailed version can be found in [8].

For a given SFC $\mathcal{S} = (X, A, S, S_0, T)$, the transformed BIP model is a composed component $\mathcal{B} = (\hat{A}, \hat{C})$, with $\hat{A} \subset L \times P \times T \times V \times D$ being the set of atomic components and $\hat{C} \subset (\hat{A} \times P) \times 2^{\hat{A} \times P}$ the set of connectors. In the following we describe the transformation of various semantic entities. In the generated BIP model, an $SFCManager$ component is introduced to enforce synchronization of them: the execution of the SFC program comprising two periodic phases.

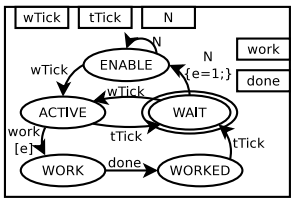

Fig. 4: BIP Atomic Component for an Action Control Block

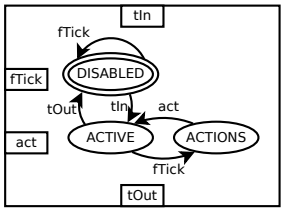

Fig. 5: BIP Atomic Component for an SFC Step

A. Atomic Components

Several transformation templates can be identified for the transformation of SFC elements into BIP elements.

Actions: Actions are described as FBDs. FBDs are translated into C code. An action BIP component is created that encapsulates this C code.

Action Control Blocks: For an action component $\hat{a} \in \hat{A}$ transformed from an SFC action $a \in A$, a special atomic component called Action Control Block (ACB) is equipped, which steers the execution of an action.

We use $\hat{b}_a$ to denote the ACB component created for $\hat{a}$ and $\hat{B}$ for the set of ACB components for all actions.

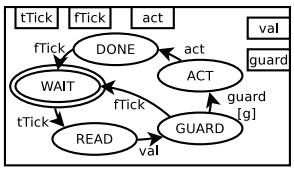

Fig. 6: BIP Atomic Component for an SFC Guard

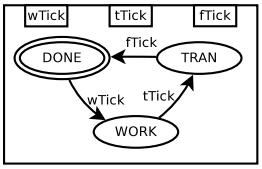

Fig. 7: BIP Atomic Component for SFC Manager

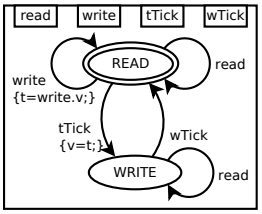

Fig. 8: BIP Atomic Component for Global Variable

Figure 4 shows a graphical representation of the ACB component. The $work$ and $done$ ports are used to drive its action component, the $xTick$ ports are used to synchronize the execution phases and the N port is used by the step components to activate the action.

Steps: A step in SFC $s \in S$ is represented by an atomic BIP Component $\hat{s} \in \hat{S}$ shown in Figure 5. The tIn and $tOut$ Ports are used to enable and disable the steps; $fTick$ is used for synchronization within the SFC evaluation phases. The act port is used to activate actions.

Guard: For each step transition $t = (t_{src}, t_g, t_{tgt}) \in T$ in SFC, a guard t_g is associated. This guard is built as an atomic component $\hat{t}_g \in \hat{A}_G$ in the BIP domain (Figure 6). The most important part of the guard is g, which represents the condition for this guard. Because the conditions may differ, a BIP guard component has to be customized for each SFC guard. Also, access to variables has to be added. The sample guard in Figure 6 just reads one variable val.

SFC Manager: The SFC Manager component, displayed in Figure 7, enforces the synchronization of the SFC execution phases by periodically enabling its $wTick$, $tTick$ and $fTick$ ports. The $wTick$ interaction starts execution of active actions, the $tTick$ interaction starts evaluation of step transitions and $fTick$ interaction starts computation of the active actions for next cycle.

Variables: The BIP language does not support global variables that are accessible to all components. Hence, an SFC variable $x \in X$ is encapsulated as an atomic BIP component representing a global variable shown in Figure 8.

Starter: This component activates the initial steps of the SFC program.

B. Transformation Steps

The following describes the steps necessary to convert an SFC program to a BIP system. All created instances of atomic components are added to $\hat{A}$.

- For each global variable $x \in X$ we create an instance of the BIP global variable atomic component $\hat{x} \in \hat{A}_X$.
- For each Action $a \in A$ we create an instance $\hat{a} \in \hat{A}_A$ of it in BIP.
- For each Action $a \in A$ we create an instance of the ACB component $\hat{b}_a \in \hat{A}_B$ in BIP and connect the $work$ and $done$ ports correspondingly.

$$\hat{C}_A := \Big\{\Big(\big(\hat{b}_a, work\big), \{(\hat{a}, work)\}\Big) \,|\hat{a} \in \hat{A}_A \wedge \hat{b}_a \in \hat{A}_B\Big\} \cup \Big\{\Big(\big(\hat{b}_a, done\big), \{(\hat{a}, done)\}\Big) \,|\hat{a} \in \hat{A}_A \wedge \hat{b}_a \in \hat{A}_B\Big\}$$

- For each SFC Step $s \in S$ we instantiate the BIP Step component $\hat{s} \in \hat{A}_S$.
- For each initial step $s \in S_0$ we create a BIP Starter instance $\hat{r}_s \in \hat{A}_{S_0}$ and create a connector to $\hat{s}$: $\hat{C}_{S_0}$.
- For each SFC transition $(t_{src}, g, t_{dst}) = t \in T$ we create a guard component $\hat{g}_t \in \hat{A}_G$ in BIP and connect it to $tOut$ of t_{src} and tIn of t_{tgt}. If the reading of variables is needed for evaluating the condition, necessary read ports, locations and transitions are created according to Figure 6.

$$\hat{C}_T := \bigcup_{(t_{src},g,t_{tgt})\in T} \Big\{\Big((\hat{g}_t, guard), \bigcup_{s\in t_{src}} \{(\hat{s}, tOut)\} \cup \bigcup_{s\in t_{tgt}} \{(\hat{s}, tIn)\}\Big)\Big\}$$

- Connect all actions and guards to global variables if necessary.

$$\hat{C}_X := \bigcup_{\substack{(\hat{x},\hat{a})\in \\ \hat{A}_X\times\hat{A}_A}} \{\big((\hat{x}, read), \{(\hat{a}, \hat{x}_r)\}\big), \big((\hat{a}, \hat{x}_w), \{(\hat{x}, write)\}\big)\} \cup \bigcup_{\substack{(\hat{x},\hat{g})\in \\ \hat{A}_X\times\hat{A}_G}} \{\big((\hat{x}, read), \{(\hat{a}, \hat{x}_r)\}\big)\}$$

- For each step $s \in S$ create a connector to the ACB $\hat{b}$ of all actions it activates.

$$\hat{C}_B := \bigcup_{s\in S} \Big\{\Big((\hat{s}, act), \bigcup_{a\in s.\Omega} \{(\hat{b}_a, N)\}\Big)\Big\}$$

- Create an instance of the SFC manager component $\hat{m}$ and connect the $wTick$, $tTick$ and $fTick$ ports to all components that expect these signal.

$$\hat{C}_M := \{((\hat{m}, wTick), \{(\hat{a}, wTick) | \hat{a} \in \hat{A}_B \cup \hat{A}_X \cup \hat{A}_{S_0}\})\} \cup \{((\hat{m}, tTick), \{(\hat{a}, tTick) | \hat{a} \in \hat{A}_B \cup \hat{A}_G \cup \hat{A}_X\})\} \cup \{((\hat{m}, fTick), \{(\hat{a}, fTick) | \hat{a} \in \hat{A}_S \cup \hat{A}_G\})\}$$

- Putting everything together :

$$\hat{A} := \hat{A}_X \cup \hat{A}_A \cup \hat{A}_B \cup \hat{A}_S \cup \hat{A}_{S_0} \cup \hat{A}_G \cup \{\hat{m}\}$$
$$\hat{C} := \hat{C}_X \cup \hat{C}_A \cup \hat{C}_{S_0} \cup \hat{C}_T \cup \hat{C}_B \cup \hat{C}_M$$
$$\mathcal{B} := (\hat{A}, \hat{C})$$

V. Invariant Preservation of the SFC to BIP Transformation

Semantics of IEC 61131–3 and BIP, invariants and safety properties do specify sets of states / configurations. Invariants are an over approximation of the semantics of a system: The state / configuration space associated with a system is a subset of the state / configuration space specified by the invariant. Safety properties can be regarded as an over approximation of invariants.

In this section, we define safety property preserving invariant transformations. These correspond to our use-cases from Section I-A. By definition, an invariant for IEC 61131–3 is a predicate on configurations that holds for all reachable configurations. This definition implies a restriction to SFC based invariants and the treatment of actions as (atomic) functions. An invariant on a BIP model is a predicate that holds for all reachable states. Given an SFC $\mathcal{S} = (X, A, S, S_0, T)$ and the transformed BIP model $\mathcal{B} = \mathcal{T}(\mathcal{S})$, we define an invariant transformation function T_I and a safety property transformation function T_R. The former takes an invariant on a BIP model and returns an invariant on an SFC model, the later takes an unsafe SFC configuration (can be translated into a desired invariant) and returns a corresponding BIP invariant.

T_I corresponds to the use-case 1 discussed in Section I-A. The function T_R corresponds to the second use-case.

Invariants on SFCs: The syntax of an SFC invariant can be expressed as follows:

$$\begin{aligned} I &::= i \wedge i \\ i &::= i' \mid i' \vee i' \mid \neg i' \\ i' &::= C \mid AS \mid AA \\ C &::= cond(X) \\ AS &::= s \in activeS \\ AA &::= a \in activeA \end{aligned}$$

Where $cond(X)$ is a predicate on the set of SFC variables X.

Invariants on BIP Models: An invariant in a BIP model can be expressed using the following syntax:

$$\begin{aligned} \hat{I} &::= i \wedge i \\ i &::= i' \mid i' \vee i' \mid \neg i' \\ i' &::= C \mid L \mid C \wedge L \\ C &::= cond(var(\hat{s})) \\ L &::= atl(\hat{s}) = l \end{aligned}$$

where $atl(\hat{s}) = l$ is true if the component $\hat{s}$ is at location l, and $cond(var(\hat{s}))$ is a predicate on the variables of $\hat{s}$. The structure of this invariant language reflects the invariant generated by D-Finder [5] and our examples from Section III.

A. Transformation of BIP Invariants to SFC Invariants

Our invariant transformation function T_I takes a BIP invariant and returns the corresponding invariant in the SFC domain. It is defined inductively:

$$\begin{aligned} T_I(i_1 \wedge i_2) &= T_I(i_1) \wedge T_I(i_2) \\ T_I(i_1 \vee i_2) &= T_I(i_1) \vee T_I(i_2) \\ &\ldots \end{aligned}$$

The function T_I keeps the structure of BIP invariants and maps each elementary BIP predicate to a corresponding predicate in the SFC domain. Given a BIP model $\mathcal{B} = \mathcal{T}(\mathcal{S})$ transformed from an SFC model $\mathcal{S}$, the mapping of elementary predicates on $\mathcal{B}$ to predicates on $\mathcal{S}$ is described using following rules. The notation $\hat{s}.L$ denotes the set of all locations of BIP atomic $\hat{s}$.

1) For a predicate on variables $p = cond(var(\hat{s}))$ we distinguish the following cases:

- if $\hat{s}$ is a global variable component created for SFC variable x, $\hat{s}$ has two local BIP variables v and t by definition. Since the relationship between v and t is a simple assignment, the condition can be written in the form $cond_t(t) \wedge cond_v(v)$. Then, the corresponding predicate in the SFC domain is $T_I(p) = cond_v(x)$, i.e. we ignore the condition on t and apply the condition on v to SFC variable x;
- if $\hat{s}$ is an ACB component, it contains a local BIP boolean variable e. The predicate p is a boolean expression on these variables. Let a be the corresponding SFC action that $\hat{s}$ is created for, the transformation of p is done by keeping the structure of expression and doing the replacement: $T_I(e) = a \in activeA$;
- if $\hat{s}$ is an other component, $T_I(p) = true$, since only global variable and ACB component contain variables according to the model transformation rules.

2) For predicates on locations we distinguish the following cases:

- if $\hat{s}$ is a step component,

$$T_I(atl(\hat{s}) = l) = \begin{cases} \neg(s \in activeS) & \text{iff } l\text{= DISABLE} \\ s \in activeS & \text{otherwise} \end{cases}$$

  where s is the step in SFC that $\hat{s}$ is transformed from;
- if $\hat{s}$ is a ACB component created for SFC action a,

$$T_I(atl(\hat{s}) = l) = \begin{cases} \bigvee_{s \in S_N(a)} s \in activeS & \text{iff } l\text{= ENABLE} \\ false & \text{otherwise} \end{cases}$$

  where S_N is the set of steps to which the action a is associated to;
- if $\hat{s}$ is another component, $T_I(p) = false$.

In the invariant transformation procedure, some unused predicates on locations are eliminated by setting them to false in the transformed SFC invariant. This is safe because the BIP component invariants have a disjunctive form as mentioned before.

Invariant Preservation: General Proof Sketch: We divide the proof of invariant preservation of the presented transformations into two steps. In the first step, we introduce a relation $R(c, \hat{c})$ between a reachable configuration c in the original SFC model and a reachable configuration $\hat{c}$ in the transformed BIP model. Our first proof goal is to show that for each SFC transition $(c, c') \in [\![SM]\!]_{SFC}$, we can always find the corresponding BIP configuration pair $(\hat{c}, \hat{c}')$ such that $R(c, \hat{c}) \wedge R(c', \hat{c}') \wedge \hat{c} \xrightarrow{\text{BM}}{}^{+}_{BIP} \hat{c}'$, where $\hat{c} \xrightarrow{\text{BM}}{}^{+}_{BIP} \hat{c}'$ is true if their exists a sequence of transitions contained in $[\![BM]\!]_{BIP}$ that transforms $\hat{c}$ to $\hat{c}'$. This property guarantees that the behavior in the SFC domain can always be traced in the BIP domain. Our proof goal in the second step is to show that the invariants are preserved for two configurations in the relation R.

The relation R puts constraints on BIP states and SFC

configurations. It has the following form:

$$\forall c = (f, activeS, activeA),\ \forall \hat{c} = (atl, \sigma),\ \ R(c, \hat{c})\ holds\ iff$$
$$rule_1(c, \hat{c}) \wedge ... \qquad where, e.g.$$
$$rule_1(c, \hat{c}) = \ \forall\ s \in S, \hat{s} \in \hat{A}_S\ .$$
$$s \in activeS \equiv \neg(atl(\hat{s}) = DISABLE)$$

The first proof goal can be divided into two subgoals $G1_a$ and $G1_b$

$$G1_a : R(c_0, \hat{c_0})$$
$$G1_b : \forall c, c', \hat{c}\ .\ c \xrightarrow{\text{SM}}_{SFC} c' \wedge R(c, \hat{c})$$
$$\rightarrow\ \exists \hat{c'}\ .\ \hat{c} \xrightarrow{\text{BM}}^{+}_{BIP} \hat{c'} \wedge R(c', \hat{c'}) \wedge BM = \mathcal{T}(SM)$$

The second step of our proof is to show that for all configuration pairs in the relation, the invariant is preserved. Formally, the second proof goal $G2$ is:

$$G2 = \forall c, \hat{c}\ .\ R(c, \hat{c})\ \rightarrow\ (\forall \hat{I}\ .\ \hat{c} \models \hat{I} \rightarrow c \models T_I(\hat{I}))$$

where $\hat{I}$ is a invariant in BIP model and T_I is the invariant transformation function that maps $\hat{I}$ to an invariant in SFC domain. The notion $\hat{c} \models \hat{I}$ means $\hat{c}$ satisfies $\hat{I}$. The proof of $G1$ and $G2$ can be conducted based on the model transformation rules and the definition of T_I. The details of the proof can be found in [8].

B. Transformation of SFC Safety Requirements into BIP Invariants

The safety property transformation function T_R is aimed for use-case 2 where we guarantee that certain safety critical configurations are unreachable. Here we define it in the following way:

$$T_R((activeS, activeA, f)) =$$
$$\forall s \in activeS\ .\ \neg(atl(\hat{s}) = DISABLE)$$
$$\wedge\ \forall a \in activeA\ .\ \hat{b}_a.e = 1$$
$$\wedge\ \forall x \in X\ .\ \sigma(\hat{x}.v) = f(x)$$

The transformation function stated above is safe because the returned invariants in the BIP domain are at most as strong as the invariants that correspond to the safety properties in the SFC domain (cf. [8]).

VI. Implementation

The IEC 61131–3 to BIP transformation is implemented in JAVA as an eclipse plug-in [1]. It takes an EasyLab model and returns the BIP code as well as an XML description of the BIP model. The generated BIP representation can be directly compiled using the BIP compiler. The EasyLab tool is extensible in sense that user-defined FBD blocks can be easily added to the component library. To sustain the extensibility, the FBD components in the BIP domain is also provided as a library. As a future work we plan to derive the BIP component library directly from the EasyLab components.

[1] *see* http://www.eclipse.org/modeling/emf/

VII. Concluding Remarks

In this paper we presented the formalization of the transformation from the SFC language of the IEC 61131–3 standard into BIP. Furthermore, we have presented transformations for invariants / safety properties between the two languages.

We are interested in extending the supported subsets with timers and more asynchronous behavior. This will go beyond the features offered by the EasyLab subset. Regarding correctness guarantees, we are working on a certificate generation and checking infrastructure similar to those described in [6], [7], [9].

Acknowledgements

This work has been supported by the European research project ACROSS under the Grant Agreement ARTEMIS-2009-1-100208.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 1995.

[2] S. Barner, M. Geisinger, Ch. Buckl, and A. Knoll. EasyLab: Model-based development of software for mechatronic systems. Mechatronic and Embedded Systems and Applications, IEEE/ASME, October 2008.

[3] N. Bauer, R. Huuck, B. Lukoschus and S. Engell. A Unifying Semantics for Sequential Function Charts. *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, 2004.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. *Software Engineering and Formal Methods*. IEEE, 2006. (SEFM'06).

[5] S. Bensalem, M. Bozga, J. Sifakis, and T-H. Nguyen. Compositional Verification for Component-Based Systems and Application. *Automated Technology for Verification and Analysis*, volume 5311 of *LNCS*, 2008. (ATVA'08).

[6] J. O. Blech. Certifying System Translations Using Higher Order Theorem Provers. *PhD-Thesis*, ISBN 3832522115, Logos-Verlag, Berlin, 2009.

[7] J. O. Blech and B. Grégoire. Certifying Compilers Using Higher Order Theorem Provers as Certificate Checkers. Formal Methods in System Design, Springer-Verlag, 2010.

[8] J. O. Blech, A. Hattendorf, and J. Huang. Towards a Property Preserving Transformation from IEC 61131-3 to BIP. eprint arXiv:1009.0817, 09/2010.

[9] J. O. Blech and M. Périn. Generating Invariant-based Certificates for Embedded Systems. *Transactions on Embedded Computing Systems*, ACM. *accepted*.

[10] S. Bornot, R. Huuck, Y. Lakhnech, B. Lukoschus. An Abstract Model for Sequential Function Charts. Discrete Event Systems: Analysis and Control, Workshop on Discrete Event Systems, 2000.

[11] S. Bornot, R. Huuck, Y. Lakhnech, B. Lukoschus. Verification of Sequential Function Charts using SMV. Parallel and Distributed Processing Techniques and Applications. CSREA Press, June 2000. (PDPTA 2000)

[12] M. Bozga, V. Sfyrla, J. Sifakis. Modeling synchronous systems in BIP. Embedded software, ACM, 2009. (EMSOFT '09)

[13] M. Y. Chkouri, M. Bozga. Prototyping of Distributed Embedded Systems Using AADL. Model Based Architecting and Construction of Embedded Systems ACES-MB, 2009.

[14] H. Ehrig and K. Ehrig. Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation Graph and Model Transformation. Elsevier Science, 2005. (GraMoT'05)

[15] M. Jaffe, N. Leveson, M. Heimdahl, and B. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 1991.

[16] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *Principles of programming languages*. ACM Press, 2006. (POPL'06).

[17] C. Loiseaux and S. Graf and J. Sifakis and A. Bouajjani and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design, 1995.

[18] G. C. Necula. Proof-carrying code. *Principles of Programming Languages*. ACM Press, 1997. (POPL'97).
[19] M. Nordio. Proofs and Proof Transformations for Object-Oriented Programs. PhD dissertation 18689, Department of Computer Science, ETH Zurich, October 2009.
[20] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. *Tools and Algorihtms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, 1998. (TACAS'98).
[21] Programmable controllers - Part 3: Programming languages, IEC 61131-3: 1993, International Electrotechnical Commission, 1993.