Automatic Generation of Provably Correct Embedded Systems

Shang-Wei Lin[1], Yang Liu[1], Pao-Ann Hsiung[3], Jun Sun[2], and Jin Song Dong[1]

[1] National University of Singapore
{tsllsw,tslliuya,dongjs.comp}@nus.edu.sg
[2] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[3] National Chung Cheng University, Chia-Yi, Taiwan
pahsiung@cs.ccu.edu.tw

Abstract. With the demand for new and complicated features, embedded systems are becoming more and more difficult to design and verify. Even if the design of a system is verified, how to guarantee the consistency between the design and its implementation remains a big issue. As a solution, we propose a framework that can help a system designer to model his or her embedded system using a high-level modeling language, verify the design of the system, and automatically generate executable software codes whose behavior semantics are consistent with that of the high-level model. We use two case studies to demonstrate the effectiveness of our framework.

1 Introduction

Embedded systems are increasingly used to control our daily life or critical tasks, which makes it important to guarantee the correctness. However, embedded systems are more and more complex due to the demand for new and complicated features, which makes it challenging to verify the correctness.

To verify embedded systems, system designers usually model the control behavior of the system using state-based diagrams such as UML state machines, where data and its operations are abstracted because modeling data and its operations by state-based diagrams is not straightforward. Furthermore, the number of possible system states often increases exponentially, which makes system verification (e.g., by model checking) infeasible. Even if data and its operations are specified by designers directly in high-level programming languages, the exact interactions among the user-written code and user-specified models are not precise, which leads to a possible semantic gap between the code and the models. During verification, if any fault occurs within this semantic gap, it will go undetected.

In this work, we propose a framework as a solution to bridge the semantic gap. Our framework adopts *communication sequential program* (CSP#) [12] as our high-level modeling facility. CSP# is a an expressive formal modeling language, which can be used to model concurrent processes communicating via both shared memory and message passing. CSP# even allows users to declare variables and model data operations on the declared variables in programming languages. With its expressiveness, the

control behavior and data operations of the system are formally and precisely modeled as a whole, which eliminates the semantic gap between control and data. In addition, to guarantee the consistency between the design of an embedded system and its implementation, the proposed framework automatically generates executable embedded software in a constructive way for the designer such that the functional correctness of the high-level CSP# models is transferred to the synthesized code.

Furthermore, functional correctness is not the only consideration of embedded systems. Non-functional requirements such as low-power consumption are also important. Our framework also provides the facility to specify non-functional requirements as assertions in CSP#, which can help designers to model and verify non-functional requirement. The contributions of this work are three-fold as listed below.

1. We propose a complete framework for modeling and verifying embedded systems.
2. The proposed framework can further automatically generate software code with functional correctness, and we have proved that the behavior semantics of the generated code conforms to the high-level model.
3. Non-functional requirements can be modeled in our framework and be verified using model checking or scheduling.

The rest of this paper is organized as follows. Section 2 gives related works. Section 3 gives an introduction to the CSP# language as well as its operational semantics. Section 4 introduces the proposed framework and proves the correctness of the generated code. We have applied our framework on two case studies, as given in Section 5. Section 6 concludes this paper and discusses some future works.

2 Related Work

Toolsets for design and verification of systems include the SCR toolset [5], NIMBUS [13], and SCADE Suite [2]. The SCR toolset uses the SPIN model checker, PVS-based TAME theorem prover, a property checker, and an invariant generator for formal verification of a real-time embedded system specified using the SCR tabular notation. It supports test case generation the TVEC toolset. NIMBUS is a specification-based prototyping framework for embedded safety-critical systems. It allows execution of software requirements expressed in RSML with various models of the environment, and it supports model checking and theorem proving. However, they do not support automatic code generation. SCADE Suite uses safe state machines (SSM) for requirement specification and automatically generates DO-178B Level A compliant and verified C/Ada code for avionics systems.

Worldwide research projects targeting embedded real-time software design include the MoBIES project [9] supported by USA DARPA, the HUGO project [7] supported by Germany's Ludwig-Maximilians-Universitä München, and the TIMES project [3] supported by the Uppsala University of Sweden. However, none of them completely support system designers to model, verify, and implement embedded systems with a comprehensive framework.

Verifiable Embedded Real-Time Application Framework (VERTAF) [6], is a comprehensive framework supporting high-level modeling, verification, and automatic code

generation for real-time embedded systems. VERTAF adopts UML diagrams as its modeling language. Since UML diagrams are informal and have limited expressiveness, the exact interactions among the user-written high-level programming language code and the UML models cannot be precisely specified in VERTAF. This leads to a possible semantic gap between the user-written code and the user-specified models. During verification, if any fault occurs within this semantic gap, it will go undetected by the model checker. In this work, we extend the modeling ability of VERTAF with the CSP# language and enhance the ability of code synthesis such that executable verified software code can be generated automatically and the generated code is consistent with the high-level models.

3 Preliminaries

This section is devoted to a brief introduction to the CSP# language and its operational semantics. A CSP# *process* is defined (using a core subset of process constructs) as,

$$P ::= Stop \mid Skip \mid e\{prog\} \rightarrow P \mid ch!x \rightarrow P \mid ch?x \rightarrow P \mid P;Q \\ \mid [b]P \mid \text{if } b \{P\} \text{ else } \{Q\} \mid P \,|||\, Q \mid P[]Q$$

where e is an event with an operational sequential program $prog$, ch is a channel with a bounded buffer size, x is a variable, and b is a Boolean expression.

Let Σ denote the set of all visible events, τ denote an invisible action, and $\checkmark$ denote the the special event of termination. Let $\Sigma_\tau = \Sigma \cup \{\tau\}$ and $\Sigma_{\tau,\checkmark} = \Sigma \cup \{\tau, \checkmark\}$. The $Stop$ process communicates nothing, also called deadlock and $Skip = \checkmark \rightarrow Stop$. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. If e is attached with a program $prog$ (also called *data operation*), the program is executed automatically together with the occurrence of the event. The attached program $prog$ can be C# program or any language with reflection that can be used for observing and/or modifying program execution at runtime. Channel communication process $ch!x \rightarrow P$ or $ch?x \rightarrow P$ behaves as P after sending or receiving x through the channel ch, respectively. Sequential composition $(P;Q)$ behaves as P until its termination and then behaves as Q. Conditional choice $if\ b\ \{P\}\ else\ \{Q\}$ behaves as P if b evaluates to true and behaves as Q otherwise. Process $[b]P$ waits until condition b becomes true and then behaves as P. Process $P \,|||\, ^1Q$ runs P and Q independently and they can communicate through shared variables. Process $P[]^2Q$ behaves as either P or Q randomly. Example 1 illustrates how to model a system using CSP#.

Example 1. Peterson's algorithm [10] was designed for the synchronization problem for processes. Each process has to get into its critical section to do some computation, but only one process is allowed in its critical at the same time. Listing 1.1 shows the CSP# model of the Peterson's algorithm for two processes P_0 and P_1, where `req.0`

[1] In this work, we only consider the *interleaving* composition ($|||$) because *parallel* composition ($||$) is not natural and practical in real programs.

[2] CSP# provides *general*, *external*, and *internal* choice compositions. In this work, we focus on the general choice composition. The syntax and semantics of the three choice compositions can be found in PAT user manual.

Listing 1.1. CSP# Model for Peterson's Algorithm

```
var turn;        var pos[2];

P0() = req.0{pos[0] = 1; turn = 1} -> Wait0(); cs.0 -> reset.0{pos[0] = 0} -> P0();
Wait0() = if (pos[1] == 1 && turn == 1) { Wait0() } else { Stop };

P1() = req.1{pos[1] = 1; turn = 0} -> Wait1(); cs.1 -> reset.1{pos[1] = 0} -> P1();
Wait1() = if (pos[0] == 1 && turn == 0) { Wait1() } else { Stop };

Peterson() = P0() ||| P1();
```

and `cs.0` represent that P_0 requests to enter its critical section and P_0 is currently in its critical section, respectively.

Given a system modeled by a CSP# process, a *system configuration* is a three-tuple (P, V, C) where P is the current process expression, V is the current valuation of the global variables which is a mapping from a name to a value, and C is the current status of channels which is a mapping from a channel name to a sequence of items in the channel. A transition is of the form $(P, V, C) \xrightarrow{e} (P', V', C')$ meaning that (P, V, C) evolves to (P', V', C') by performing event e. The meaning or behavior of a CSP# process can be described by the operational semantics, as shown in Fig 1, where $e \in \Sigma$ and $e_\tau \in \Sigma_\tau$. We use $upd(V, prog)$ to denote the function which, given a sequential program $prog$ and V, returns the modified valuation function V' according to the semantics of the program. We use $V \models b$ (or $V \not\models b$) to denote that boolean condition b evaluates to true (or false) given V. We use $eva(V, exp)$ to denote the value of the expression evaluated with variable valuations in V. By the operational semantics, a CSP# process is associated with a labeled transition systems (LTS), as formulated in Definition 1.

Definition 1. ***(Labeled Transition System).*** *A* labeled transition system *(LTS) is a 4-tuple $(S, \Sigma_{\tau,\surd}, \longrightarrow, s_0)$ where S is a set of system configurations, $\Sigma_{\tau,\surd}$ is a set of events, $\longrightarrow \subseteq S \times \Sigma_{\tau,\surd} \times S$ is a transition relation, and $s_0 \in S$ is the initial state. We use $s \xrightarrow{\alpha} s'$, for simplicity, to denote $(s, \alpha, s') \in \longrightarrow$ where $s, s' \in S$ and $\alpha \in \Sigma_{\tau,\surd}$.*

4 Design and Synthesis Flow

Fig. 2 shows the overall flow of our approach. It consists of two main phases, namely, design phase and synthesis phase, and we refer to them as the front-end and back-end, respectively. The front-end is further divided into three subphases, namely, modeling, scheduling, and verification phases. There are two subphases in the back-end, namely, hardware mapping and code generation phases. The details of each phase are described in the following subsections.

4.1 Modeling

In the modeling phase, we adopt *communicating sequential programs* (CSP#) [12] as our modeling language, which is a high-level formal modeling language for concurrent

$$\frac{}{(Skip, V, C) \xrightarrow{\checkmark} (Stop, V, C)}\ [\text{skip}] \qquad \frac{(P, V, C) \xrightarrow{e} (P', V', C'), V \models b}{([b]P, V, C) \xrightarrow{e} (P, V, C)}\ [\text{guard}]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C)}{(P \,|||\, Q, V, C) \xrightarrow{e} (P' \,|||\, Q, V', C)}\ [\text{int1}] \qquad \frac{(Q, V, C) \xrightarrow{e} (Q', V', C)}{(P \,|||\, Q, V, C) \xrightarrow{e} (P \,|||\, Q', V', C)}\ [\text{int2}]$$

$$\frac{(P, V, C) \xrightarrow{\checkmark} (P', V', C) \text{ and } (Q, V, C) \xrightarrow{\checkmark} (Q', V', C)}{(P \,|||\, Q, V, C) \xrightarrow{\checkmark} (P' \,|||\, Q', V, C)}\ [\text{int3}]$$

$$\frac{}{(e\{prog\} \rightarrow P, V, C) \xrightarrow{e} (P, upd(V, prog), C)}\ [\text{prog}]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C)}{(P\ ;\ Q, V, C) \xrightarrow{e} (P'\ ;\ Q, V', C)}\ [\text{seq1}] \qquad \frac{(P, V, C) \xrightarrow{\checkmark} (P', V', C)}{(P\ ;\ Q, V, C) \xrightarrow{\tau} (Q, V', C)}\ [\text{seq2}]$$

$$\frac{V \models b}{(if\ b\ \{P\}\ else\ \{Q\}, V, C) \xrightarrow{\tau} (P, V, C)}\ [\text{cond1}]$$

$$\frac{V \not\models b}{(if\ b\ \{P\}\ else\ \{Q\}, V, C) \xrightarrow{\tau} (Q, V, C)}\ [\text{cond2}]$$

$$\frac{(P, V, C) \xrightarrow{e_\tau} (P', V', C)}{(P\ []\ Q, V, C) \xrightarrow{e_\tau} (P', V', C)}\ [\text{ch1}] \qquad \frac{(Q, V, C) \xrightarrow{e_\tau} (Q', V', C)}{(P\ []\ Q, V, C) \xrightarrow{e_\tau} (Q', V', C)}\ [\text{ch2}]$$

$$\frac{C(ch) \text{ is not empty}}{(ch?x \rightarrow P, V, C) \xrightarrow{ch?C(ch).head} (P, V, C'), \text{where } C'(ch) = C(ch) \setminus \{C(ch).head\}}\ [\text{in}]$$

$$\frac{C(ch) \text{ is not full}}{(ch!exp \rightarrow P, V, C) \xrightarrow{ch!eva(V,exp)} (P, V, C'), \text{where } C'(ch) = C(ch) \cup \{eva(V, exp)\}}\ [\text{out}]$$

Fig. 1. Operational Semantics of CSP# Processes

systems. In CSP#, complex communications such as shared memory and message passing can be easily modeled, and system designers can even include code fragments in the model. Most importantly, a complete simulation and verification tool support, *Process Analysis Toolkit* (PAT) [8], is available.

To model the interactions between system model and hardware, we classify hardware components and define generic hardware API for each class, e.g., for multi-media hardware components such as a LCD, we define `init()`, `reset()`, `display()`, and `refresh()`. Designers can use the generic hardware APIs as data operations in CSP# to describe the system behavior of interacting with hardware. Note that we assume hardware APIs have no side effects, i.e., they do not change the values of the declared variables in the model. Thus, they are not considered in the verification.

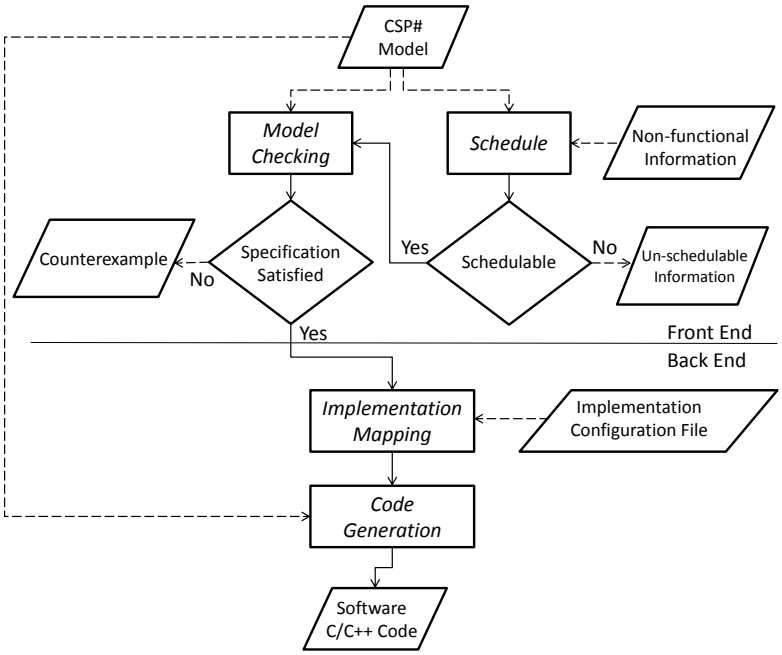


Fig. 2. Overall Flow

4.2 Scheduling

Non-functional requirements such as low-power consumption and worst-case execution time can be evaluated in this phase. To evaluate the non-functional parts of systems, the system designer can describe the power consumption or execution time of each event and each data operation (obtained by profiling) in the *non-functional information* file, and the requirements of non-functional properties are specified as assertions in CSP# in the modeling phase. Our framework provides several scheduling algorithms, such as *Quasi-dynamic Scheduling* (QDS) [6], to evaluate the non-functional properties. If the system design is not feasible, information on why it is non-schedulable will be given to designers. If the system design is schedulable, the flow goes to the verification phase.

4.3 Formal Verification

To check the functional correctness of the systems, we apply *model checking* [4] in this phase. Model checking is an automatic analysis procedure that can show if a system satisfies a temporal property or violates it with a counterexample. We integrate the PAT [8] model checker, which takes CSP# model as its input, with our framework. PAT is a self-contained model checker to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains. It implements various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement

checking and probabilistic model checking. If the system model does not satisfy the assertions of functional correctness, a counterexample will be given to the system designer. If the assertions are all satisfied, then the flow moves on to the back-end phase.

4.4 Implementation Mapping

To make the design of a system executable on a real hardware platform, every generic hardware API used in the modeling phase has to be mapped into a concrete implementation. In this phase, the target hardware platform where the final system runs is configured, and each generic hardware API is mapped into its corresponding implementation in the specified hardware platform. Hardware-dependent files such as make files and header files are all included in this phase.

4.5 Automatic Code Generation

In the code generation phase, executable software code is automatically generated from the high-level CSP# model in two steps. The CSP# processes are first translated into state machine models and then software code is synthesized from the translated state machines. The reasons for the two-phase synthesis instead of generating software code directing from CSP# model are as follows: (1) state machines are still the most popular models in industries, and the two-phase synthesis gives designers the flexibility to exchange state machine models of their system designs. (2) there is a mature middleware library, *Quantum Platform* (QP) [11], providing a programming paradigm for implementing the state machine models in C/C++ programming language. In the paradigm supported by QP, a state and a transition have their corresponding implementation patterns such that general principles can be concluded and code synthesis for state machines can be automated. In addition, QP supports many operating systems such as Linux/BSD, Windows/WinCE, μC/OS-II, etc., and many hardware platforms such as 80X86, ARM-Cortex/ARM9/ARM7, etc, which makes the generated software codes portable. Fig. 3 shows the architecture of QP.

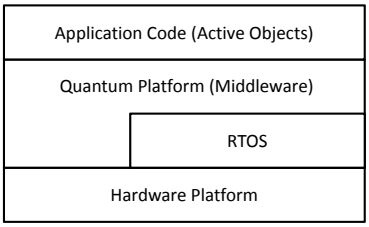


Fig. 3. Multi-Layer Code Generation

We formulate a state machine in Definition 2 and the interleaving, sequential, and choice compositions between two state machines in Definitions 3 to 5, respectively.

Definition 2. ***(State Machine)****. A state machine is a* 6*-tuple* $\mathcal{M} = (S, \Sigma, B, A, s_0, T)$ *where* S *is a set of states;* $s_0 \in S$ *is the initial state;* Σ *is a set of events;* B *is a set of Boolean expressions;* A *is a set of actions;* $T \subseteq S \times (\Sigma \cup \{\tau, \checkmark\}) \times B \times A^* \times S$ *is a transition relation. We use* $s \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} s'$ *to denote a transition, where* $s \in S$ *is the source state,* $s' \in S$ *is the destination state,* $e \in \Sigma$ *is the event trigger,* $b \in B$ *is the triggering condition, and* $(a_1; a_2; \cdots; a_n) \in A^*$ *is a sequence of actions for* $a_i \in A$ *and* $i \in \{1, 2, \ldots, n\}$*. We define* $Post(s, e) = \{s' \in S \mid s \xrightarrow{e/} s'\}$.

Definition 3. ***(Interleaving)****. Given two state machines* $\mathcal{M}_i = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *for* $i \in \{1, 2\}$*, the* interleave composition *is the state machine* $\mathcal{M}_1 \,|||\, \mathcal{M}_2 = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, B_1 \cup B_2, A_1 \cup A_2, s_0^1 \times s_0^2, T)$ *where* T *is defined as follows:*

$$
\begin{array}{ll}
(s_1, s_2) \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} (s_1', s_2) & \text{if } s_1 \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} s_1' \\
(s_1, s_2) \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} (s_1, s_2') & \text{if } s_2 \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} s_2' \\
(s_1, s_2) \xrightarrow{\checkmark/} (s_1', s_2') & \text{if } s_1 \xrightarrow{\checkmark/} s_1' \text{ and } s_2 \xrightarrow{\checkmark/} s_2'
\end{array}
$$

Definition 4. ***(Sequential Composition)****.*
Given two state machines $\mathcal{M}_i = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *for* $i \in \{1, 2\}$*, the* sequential composition *is the state machine* $(\mathcal{M}_1; \mathcal{M}_2) = (S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, B_1 \cup B_2, A_1 \cup A_2, s_0^1, T)$ *where* T *is defined as follows, where* $s \in S_1$.

$$T = T_1 \cup T_2 \setminus \{s \xrightarrow{\checkmark/} s' \in T_1 \mid s' \in Post(s, \checkmark)\} \cup \{s \xrightarrow{\tau/} s_0^2 \mid Post(s, \checkmark) \neq \emptyset\}.$$

Definition 5. ***(Choice Composition)****.*
Given two state machines $\mathcal{M}_i = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *for* $i \in \{1, 2\}$*, the* choice composition *is the state machine* $(\mathcal{M}_1 [] \mathcal{M}_2) = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *where* i *is randomly chosen from* $\{1, 2\}$.

Fig. 4 shows the one-to-one mapping from CSP# processes into state machines. The translation is performed constructively according the mapping step by step for each CSP# process. The translated state machine $\mathcal{M}_0$ for process P_0 in Example 1 is shown in Fig. 5. We omit the translated state machine $\mathcal{M}_1$ for process P_1 here since it is the same as M_0 except the conditions on transitions are symmetric to those of $\mathcal{M}_0$.

Theorem 1 proves that the behavior of the translated state machines is consistent with the behavior of the original CSP# models based on the concept of bisimulation.

Definition 6. ***(Bisimulation)****. Given two LTS* $L_i = (S_i, \Sigma, \longrightarrow_i, s_0^i)$ *for* $i \in \{1, 2\}$*, we say two states* $p \in S_1$ *and* $q \in S_2$ *are* bisimulation *of each other, denoted by* $p \approx q$ *iff*

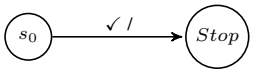
- *for all* $\alpha \in \Sigma$ *if* $p \xrightarrow{\alpha}_1 p'$*, then there exists* q' *such that* $q \xrightarrow{\alpha}_2 q'$ *and* $p' \approx q'$
- *for all* $\alpha \in \Sigma$ *if* $q \xrightarrow{\alpha}_2 q'$*, then there exists* p' *such that* $p \xrightarrow{\alpha}_1 p'$ *and* $p' \approx q'$

We say L_1 *and* L_2 *are* bisimulation *of each other, denoted by* $L_1 \approx L_2$ *iff* $s_0^1 \approx s_0^2$.

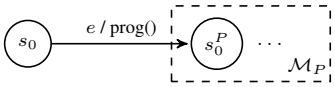
Theorem 1. *The translated state machine is a bisimulation of the original CSP# model.*

(a) $Stop$

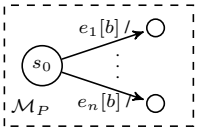
(b) $Skip$



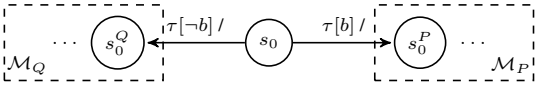
(c): $e\{prog\} \rightarrow P$



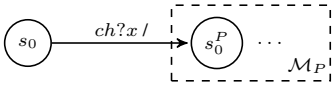
(d) $[b]P$



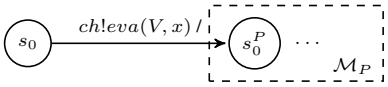
(e) $if\ b\ \{P\}\ else\ \{Q\}$



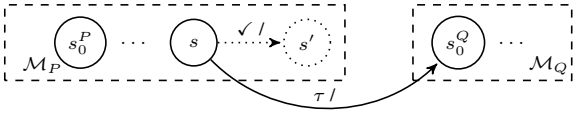
(f) $ch?x \rightarrow P$



(g) $ch!x \rightarrow P$



(h) $P;Q$



(h) $P[]Q$

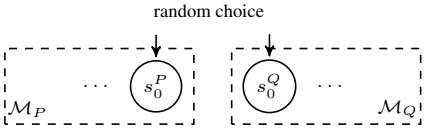


Fig. 4. Translation Rules from CSP# Processes and State Machines

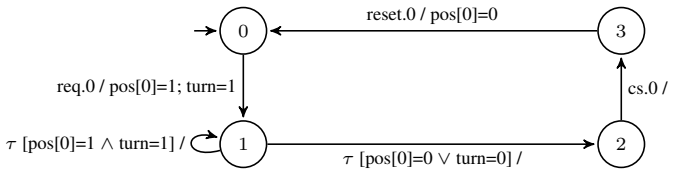


Fig. 5. Generated State Machine $\mathcal{M}_0$ for Process P_0

Proof. Given an CSP# process expression $\mathcal{E}$, let the labeled transition system associated with the process be $L_{\mathcal{E}} = (O, \Sigma_{\tau,\checkmark}, \longrightarrow_1, o_0)$. Let the translated state machine w.r.t. the CSP# process be $\mathcal{M}_{\mathcal{E}}$ and its associated labeled transition system be $L_{\mathcal{M}_{\mathcal{E}}} = (S, \Sigma_{\tau,\checkmark}, \longrightarrow_2, s_0)$. We want to prove that $L_{\mathcal{E}} \approx L_{\mathcal{M}_{\mathcal{E}}}$, i.e., $o_0 \approx s_0$. We use $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$ to denote $L_{\mathcal{E}} \approx L_{\mathcal{M}_{\mathcal{E}}}$. It can be proved by a structural induction on the CSP# expression from the following primitive processes.

- $Stop$: By CSP# operational semantics, there is no transition rule for the $Skip$ process, so L_{Stop} is a LTS having a single state without any transitions. Its corresponding state machine $\mathcal{M}_{Stop}$ also has one state without any transitions, as shown in Fig. 4 (a). Thus, $Stop \approx \mathcal{M}_{Stop}$.
- $Skip$: By CSP# operational semantics, $Skip = \checkmark \rightarrow Stop$. the LTS $L_{\mathcal{E}}$ has two states o_0, o and one transition $o_0 \xrightarrow{\checkmark} o$, where $o_0 = (Skip, V, C)$ and $o = (Stop, V, C)$. The translated state machine $\mathcal{M}_{\mathcal{E}}$ has two states s_0, s and one transition $s_0 \xrightarrow{\checkmark/} s$, as shown in Fig. 4 (b). It is obvious $o_0 \approx s_0$. Thus, $Skip \approx \mathcal{M}_{Skip}$.
- $\mathcal{E} = e\{prog\} \rightarrow P$: By CSP# operational semantics, the LTS $L_{\mathcal{E}}$ takes transition $o_0 \xrightarrow{e} o$ and behaves like P, where $o_0 = (e\{prog\} \rightarrow P, V, C)$ and $o = (P, upd(V, prog), C)$. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. We add transition $s_0 \xrightarrow{e/prog()} s_0^P$ in the translated state machine $\mathcal{M}_{\mathcal{E}}$, as shown in Fig. 4 (c), where s_0^P is the initial state of $\mathcal{M}_P$. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = [b]P$: By CSP# operational semantics, process $[b]P$ behaves like P only if b holds, i.e., process P can perform an event only if b holds. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. For each outgoing transition from the initial state of $\mathcal{M}_P$, We add a triggering condition b, as shown in Fig. 4 (d), which guarantees that each outgoing transition from the initial state is taken only if b holds. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = if\ b\ \{P\}\ else\ \{Q\}$: By CSP# operational semantics, the LTS $L_{\mathcal{E}}$ may take transition $o_0 \xrightarrow{\tau} o'$ if b holds; otherwise it takes transition $o_0 \xrightarrow{\tau} o''$, where $o_0 = (if\ b\ \{P\}\ else\ \{Q\}, V, C)$, $o' = (P, V, C)$, and $o'' = (Q, V, C)$. Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the translated state machines w.r.t. P and Q, respectively, and $P \approx \mathcal{M}_P$, $Q \approx \mathcal{M}_Q$. In the initial state s_0 of $\mathcal{M}_{\mathcal{E}}$, two outgoing transitions $s_0 \xrightarrow{\tau[b]/} s_0^P$ and $s_0 \xrightarrow{\tau[\neg b]/} s_0^Q$ are available, as shown in Fig. 4 (e), where s_0^P and s_0^Q are the initial states of $\mathcal{M}_P$ and $\mathcal{M}_Q$, respectively. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.

– $\mathcal{E} = ch?x \rightarrow P$: By CSP# operational semantics, if the channel ch is not empty, the LTS $L_{\mathcal{E}}$ takes transition $o_0 \xrightarrow{ch?C(ch).head} o$, where $o_0 = (ch?x \rightarrow P, V, C)$, $o = (P, V, C')$, and $C'(ch) = C(ch) \backslash \{C(ch).head\}$, and behaves like P. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. In the initial state s_0 of $\mathcal{M}_{\mathcal{E}}$, we add transitions $s_0 \xrightarrow{ch?C(ch).head/} s_0^P$, as shown in Fig. 4 (f), where s_0^P is the initial state of $\mathcal{M}_P$. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
– $\mathcal{E} = ch!x \rightarrow P$: By CSP# operational semantics, if the channel ch is not full, the LTS $L_{\mathcal{E}}$ takes transition $o_0 \xrightarrow{ch!eva(V,x)} o$, where $o_0 = (ch!x \rightarrow P, V, C)$, $o = (P, V, C')$, and $C'(ch) = C(ch) \cup \{eva(V, exp)\}$, and then behaves like P. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. In the initial state s_0 of $\mathcal{M}_{\mathcal{E}}$, we add transitions $s_0 \xrightarrow{ch!eva(V,x)/} s_0^P$, as shown in Fig. 4 (g), where s_0^P is the initial state of $\mathcal{M}_P$. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
– $\mathcal{E} = P \,|||\, Q$: By CSP# operational semantics, the transitions available in $L_{\mathcal{E}}$ are

$$(P \,|||\, Q, V, C) \xrightarrow{e} (P' \,|||\, Q, V', C) \quad \text{if } (P, V, C) \xrightarrow{e} (P', V', C), e \neq \checkmark$$
$$(P \,|||\, Q, V, C) \xrightarrow{e} (P \,|||\, Q', V', C) \quad \text{if } (Q, V, C) \xrightarrow{e} (Q', V', C), e \neq \checkmark$$
$$(P \,|||\, Q, V, C) \xrightarrow{\checkmark} (P' \,|||\, Q', V', C) \quad \text{if } \begin{cases} (Q, V, C) \xrightarrow{\checkmark} (Q', V', C) \\ (P, V, C) \xrightarrow{\checkmark} (P', V, C) \end{cases}$$

  Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the two translated state machines w.r.t. P and Q, respectively, such that $P \approx \mathcal{M}_P$ and $Q \approx \mathcal{M}_Q$. The state machine $\mathcal{M}_{\mathcal{E}}$ w.r.t. $\mathcal{E}$ is defined as $\mathcal{M}_{\mathcal{E}} = \mathcal{M}_P \,|||\, \mathcal{M}_Q$. By Definition 3, the only three transitions in $\mathcal{M}_{\mathcal{E}}$ correspond to the above three transitions in $L_{\mathcal{E}}$, respectively. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
– $\mathcal{E} = P; Q$: By CSP# operational semantics, process $\mathcal{E}$ first behaves like P until P's termination and then behaves like Q. Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the two translated state machines w.r.t. P and Q, respectively, such that $P \approx \mathcal{M}_P$ and $Q \approx \mathcal{M}_Q$. The translated state machine $\mathcal{M}_{\mathcal{E}}$, as shown in Fig. 4 (f), first behaves like $\mathcal{M}_P$. Right before the termination of $\mathcal{M}_P$, it takes transition $s \xrightarrow{\tau/} s_0^Q$ and then behaves like $\mathcal{M}_Q$, which corresponds to the operation semantics of $\mathcal{E}$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
– $\mathcal{E} = P[]Q$: By CSP# operational semantics, process $\mathcal{E}$ behaves like either P or Q randomly. Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the two translated state machines w.r.t. P and Q, respectively, such that $P \approx \mathcal{M}_P$ and $Q \approx \mathcal{M}_Q$. The translated state machine $\mathcal{M}_{\mathcal{E}}$, as shown in Fig. 4 (i), behaves like either $\mathcal{M}_P$ or $\mathcal{M}_Q$ randomly, which corresponds to the operation semantics of $\mathcal{E}$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.

Since any CSP# process expression $\mathcal{E}$ is composed with primitive processes inductively and we have proved that for each primitive process, the translated state machine is a bisimulation of it, therefore we can conclude that $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$ for any CSP# process. □

The executable software code is then synthesized from the translated state machine models. Let us take the Peterson's algorithm in Example 1 to illustrate how software code is automatically generated. In QP, an active object consists of a logical state machine structure, an event queue, and an execution thread, as shown in Fig. 6. A state machine is implemented as an active object by inheriting from the base class, `QActive`,

Listing 1.2. P0.h

```
1  class P0 : public QActive {
2     private: static QState initial(P0 *me, QEvent const *e);
3              static QState P0_0(P0 *me, QEvent const *e);
4              static QState P0_1(P0 *me, QEvent const *e);
5              static QState P0_2(P0 *me, QEvent const *e);
6              static QState P0_3(P0 *me, QEvent const *e);
7              QTimeEvt m_timeEvt;
8     public:  P0(): QActive((QStateHandler)&P0::initial), m_timeEvt(TIMEOUT_SIG){};
9              ~P0(){};
10             void req_0()        {       std::cout<<"req_0"<<std::endl;   }
11             void cs_0()         {       std::cout<<"cs_0"<<std::endl;    }
12             void reset_0()      {       std::cout<<"reset_0"<<std::endl;         }
13 };
```

provided by QP, and each state is implemented as a member function. Listing 1.2 shows the declaration file of state machine $\mathcal{M}_0$ in Fig. 5, where the four states 0, 1, 2, 3 are implemented as four member functions `P0_0()`, `P0_1()`, `P0_2()`, and `P0_3()`, respectively. Each execution thread of an active object is responsible for dispatching events in its event queue, i.e., invoking the member function representing the current state and passing the event as the argument. A transition from state s to s' in the state machine is implemented by invoking the `Q_TRAN(`s'`)` macro provided by QP.

An active object can communicate with others via shared variables and message passing. QP provides a *publish-subscription* mechanism for supporting message passing communication among active objects. Once an active object publishes an event, QP delivers the event to the event queue of whoever subscribes it, and each subscriber will receive the event by the execution thread. The dotted arrows in Fig. 6 show the paths of event passing between active objects and QP.

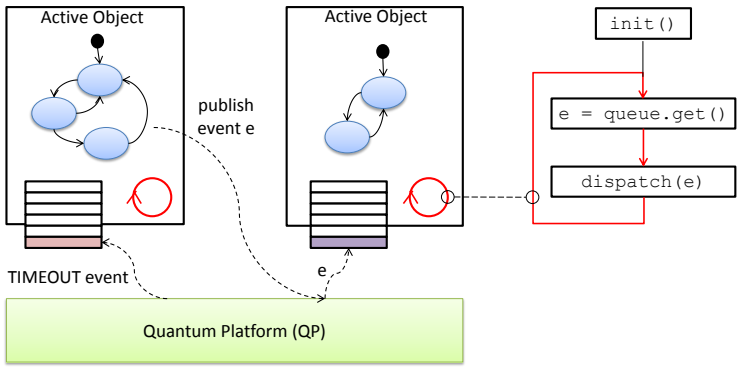


Fig. 6. Communications among Active Objects

QP also provides the timer facility such that an active object can register a time-out event of a certain time interval. After the time interval since the timer is fired, QP

Listing 1.3. P0.cpp

```
1  QState P0::initial(P0 *me, QEvent const *){      return Q_TRAN(&P0::P0_0);      }
2
3  QState P0::P0_0(P0 *me, QEvent const *e){
4     switch(e->sig){
5        case Q_ENTRY_SIG:
6           me->m_timeEvt.postIn(me, WAIT_TIME);    return Q_HANDLED();
7        case TIMEOUT_SIG:
8           me->req_0(); pos[0]=1; turn = 1;
9           return Q_TRAN(&P0::P0_1);
10    }  return Q_SUPER(&QHsm::top);
11 }
12
13 QState P0::P0_1(P0 *me, QEvent const *e){
14    switch(e->sig){
15       case Q_ENTRY_SIG:
16          me->m_timeEvt.postIn(me, WAIT_TIME);    return Q_HANDLED();
17       case TIMEOUT_SIG:
18          if(pos[1] == 1 && turn == 1) { return Q_TRAN(&P0::P0_1); }
19          else { return Q_TRAN(&P0::P0_2); }
20    }  return Q_SUPER(&QHsm::top);
21 }
22                  ...
```

puts a timeout event into the event queue of the active object that registered it. Let us recall $\mathcal{M}_0$ in Fig. 5, we implement state 0 by registering a timeout event (Line 6 in Listing 1.3). After the timeout event occurs, the active object performs the `req_0()` function representing the CSP# event `req.0` and assigns value 1 to both of the variables `pos[0]` and `turn`, which corresponds to Line 8 in Listing 1.3. Then it transits to state 1 by invoking the `Q_TRAN` macro provided by QP (Line 9).

Theorem 2 proves that the behavior of implementation in active objects conforms to the behavior of the state machines.

Theorem 2. *The behavior of the implementation conforms to the state machine.*

Proof. We give our proof sketch here. To prove that for each state machine, the behavior of its implementation in active object conforms to the original one, let us recall and analyze what the execution thread of each active object does, as shown at the right side of Fig 6. The execution thread keeps doing the followings: if there is an event in the event queue, it removes the event and dispatches the event by invoking the member function representing the current state and passing the event as the argument. In the current active state, the pointer to the member function representing the current active state is changed to its successor state by invoking `Q_TRAN()` macro provided by QP. For each transition $A \xrightarrow{e[b]/act()} B$ from state A to state B, the call graph for QP functions and member functions representing states A and B is as shown in Fig. 7, which satisfies the operational semantics of state machines. Since the implementation for each transition satisfies the operational semantics of state machines, we can conclude that the implementation for the whole active object conforms to the original state machine.

In CSP# operational semantics, the execution of each transition is atomic. If we want to conclude that two-phase code generation is sound by Theorems 1 and 2, we have to make an assumption that the execution of the action on a transition of a state

Listing 1.4. CSP# Model for the Entrance Guard System

```
#define RESET 77;
var pin[4];      var result = 0;          var open = 0;
var alarm = 0;   var symbol = 0;
channel pw 0;    channel i2ctr 0;         channel ctr2db 0;        channel db2ctr 0;
channel i2d 0;   channel ctr2a 0;         channel ctr2alm 0;       channel flag 0;

User() = User2()[]User1(); flag?x-> if(open == 1) { enter-> Skip } else { User() };
User1() = pw!1 -> pw!1 -> pw!1 -> pw!1 -> Skip;
User2() = pw!2 -> pw!RESET -> pw!2 -> pw!2 -> pw!2 -> pw!2 -> Skip;
Input() = pw?x -> check{symbol = x} -> if(symbol == RESET) { Input() } else {
   input1 { pin[0] = symbol } -> i2d!symbol -> pw?y -> check{ symbol = y } ->
   if(symbol == RESET) { Input() } else {
      input2{ pin[1] = symbol } -> i2d!symbol -> pw?z ->check{ symbol = z } ->
      if(symbol == RESET){ Input() } else {
         input3 { pin[2] = symbol } -> i2d!symbol-> pw?k -> check{symbol = k} ->
         if(symbol == RESET) { Input() } else{ input4{ pin[3] = symbol } ->
         i2d!symbol -> i2ctr!1 -> Input() }
      }
   }   };
Display() = i2d?x -> show { display('*') } -> Display();
Controller() = i2ctr?x -> ctr2db!x -> db2ctr?x ->
   if(result ==1) { ctr2a!1 -> flag!1 -> Controller() }
   else { ctr2alm!1 -> flag!1 -> Controller() };
DBMS() = ctr2db?x ->
   if(pin[0]==1 && pin[1]==1 && pin[2]==1 && pin[3]==1) {
      checkOK{result = 1;}->  db2ctr!1 -> DBMS()
   } else { checkFail{result = 0;} -> db2ctr!1 -> DBMS() };
Alarm() = ctr2alm?x -> alarmon{alarm = 1} -> alarmoff{alarm = 0} -> Alarm();
Actuator() = ctr2a?x -> opendoor{open = 1;} -> closedoor{open = 0} -> Actuator();
System() = User() ||| Input() ||| Controller() ||| DBMS() |||
           Display() ||| Actuator() ||| Alarm();
#define pre pin[0]==1 && pin[1]==1 && pin[2]==1 && pin[3]==1;
#assert System() |= [] pre -> <> open == 1;
#assert System() reaches (result == 0 && open == 1);
#assert System() |= [] result == 0 -> <> alarm == 1;
```

machine is also atomic. Fortunately, this can be guaranteed by taking and releasing a mutex at the beginning and the end of the action, respectively, which is exactly how we implement in our framework. Thus, the code generation in our framework is sound. □

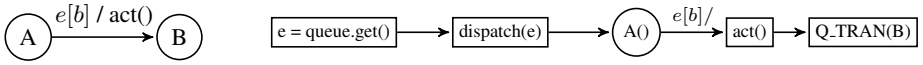


Fig. 7. Call Graph of A State Transition

5 Case Studies

We have applied the proposed framework on two case studies, namely an entrance guard system and a secure communication box. Both of the systems are modeled using the CSP# language and verified by the PAT model checker, and executable software codes for both systems are generated by our framework automatically.

Listing 1.5. CSP# Model for the Secure Communication Box

```
1 #define UserConnect 1;          #define Data 2;          #define UserDisconnect 3;
2 channel network 0;              var packet;
3
4 User() = network!UserConnect -> network!Data -> network!UserDisconnect -> User();
5 Box() = poweron -> init -> network?UserConnect -> Connected(); Box();
6 Connected() = network?x -> store{packet=x} ->
7                 if(packet == Data) {Connected()} else { reset -> Skip };
8 System() = User() ||| Box();
```

The entrance guard system (EGS) controls the entrance of a building and it consists of six components, namely Input, Display, Controller, DBMS, Actuator, and Alarm, which are modeled as six CSP# processes in Listing 1.4. Input is a keypad receiving the 4-digit password as user input. Once a user enters a digit, it saves the digit to the `PIN` array and sends the digit to Display via the channel `i2d`. If the user presses the reset button, Input collects the 4-digit password from the first digit. After receiving four digits, it sends the password to Controller via the channel `i2ctr` (Lines 10-19). Display receives digits from Input and prints a '*' symbol by calling the hardware API `display()` for each digit on the LCD (Line 20). Controller sends the query of password to DBMS and receives the result via the channel `ctr2db` (Line 21). If the password is correct, it notifies Actuator to open the door via the channel `ctr2a` (Line 22); otherwise, it notifies Alarm via the channel `ctr2alm` (Line 23). Actuator will open the door if Controller notifies it, and then it closes the door after a certain time interval (Line 29).

For verifying EGS, we add a User process to model user behavior (Line 7). The system is the interleaving of six components and User. Note that User is just used for verification, and we don't generate code for user behavior (our framework gives designers the flexibility to choose the components to generate software code for). We have verified three assertions: (1) the door never opens if the password is incorrect, (2) the door will eventually open once the correct password is entered, and (3) if the password is incorrect, Alarm will eventually be switched on (Lines 32-35). After the verification, EGS satisfies the three assertions. The detailed model and the generated software code for EGS can be found in [1].

The secure communication box (SCB) is a device for providing secure communication between two clients. SCB is funded by Singapore Defense, and it is confidential. Thus, we only give a prototype here. Listing 1.5 shows the CSP# model for the SCB prototype. After being powered on, SCB is initialized and then waits for a user connection (Line 5). In our framework, we provide the flexibility of implementing channels as interprocess or socket communications. Designers can choose the implementation for each channel. In SCB, the channel `network` is implemented as a socket communication. Once a user connects to SCB, it keeps receiving packets until the user sends a disconnection packet (Line 7). The detailed model and the generated software code can be found in [1].

6 Conclusion and Future Work

In this work, we proposed a framework that can help a system designer to model embedded systems from a high-level modeling language, verify the design of the system, and automatically generate executable software code whose behavior semantics is consistent with the high-level model. With our framework, the development cycle of embedded systems can be significantly reduced. In our future work, we plan to enhance the code generation such that the synthesized software code can be executed on multi-core architectures. We also plan to extend the syntax of the CSP# modeling language as well as its semantics such that system designers can design more featured systems such as real-time systems, probabilistic systems, safety-critical systems, and security protocols.

References

1. https://sites.google.com/site/shangweilin/pat-codegen.
2. http://www.esterel-technologies.com/products/scade-suite/.
3. T. Amnell, L. Fersman, E. Mokrushin, P. Petterson, and W. Yi. TIMES: A tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, pages 60–72, 2003.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
5. C. L. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *CAV 1998*, volume 1427, pages 526–531, 1998.
6. P. A. Hsiung, S. W. Lin, C. H. Tseng, T. Y. Lee, J. M. Fu, and W. B. See. VERTAF: An application framework for the design and verification of embedded real-time software. *IEEE Transactions on Software Engineering*, 30(10):656–674, 2004.
7. A. Knapp, S. Merz, and C. Rauh. Model checking timed uml state machines and collaboration. In *FTRTFT*, pages 395–414, 2002.
8. Y. Liu, J. Sun, and J. S. Dong. Developing model checkers using PAT. In *ATVA 2010*, volume 6252, pages 371–377, 2010.
9. D. Niz and R. Rajkumar. Time Weaver: A software-through-models framework for embedded real-time systems. In *LCTES*, pages 133–143, 2003.
10. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 10(3):115–116, 1981.
11. M. Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. Newnes, 2008.
12. J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In *TASE 2009*, volume 962, pages 127–135, 2009.
13. J. M. Thompson, M. P. E. Heimdahl, and S. P. Miller. Specification-based prototyping for embedded systems. In *SIGSOFT 1999*, pages 163–179, 1999.