KBX: Verified Model Synchronization via Formal Bidirectional Transformation

JIANHONG ZHAO, YONGWANG ZHAO*, PEISEN YAO, and FANLANG ZENG, Zhejiang University, China
BOHUA ZHAN, Institute of Software Chinese Academy of Sciences, China
KUI REN, Zhejiang University, China

Complex safety-critical systems require multiple models for a comprehensive description, resulting in error-prone development and laborious verification. Bidirectional transformation (BX) is an approach to automatically synchronizing these models. However, existing BX frameworks lack formal verification to enforce these models' consistency rigorously. This paper introduces KBX, a formal bidirectional transformation framework for verified model synchronization. First, we present a matching logic-based BX model, providing a logical foundation for constructing BX definitions within the $\mathbb{K}$ framework. Second, we propose algorithms to synthesize formal BX definitions from unidirectional ones, which allows developers to focus on crafting the unidirectional definitions while disregarding the reverse direction and missing information recovery for synchronization. Afterward, we harness $\mathbb{K}$ to generate a formal synchronizer from the synthesized definitions for consistency maintenance and verification. To evaluate the effectiveness of KBX, we conduct a comparative analysis against existing BX frameworks. Furthermore, we demonstrate the application of KBX in constructing a BX between UML and HCSP for real-world scenarios, showcasing an 72% reduction in BX development effort compared to manual specification writing in $\mathbb{K}$.

CCS Concepts: • **Computing methodologies → Modeling methodologies**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Bidirectional Transformation, Formal Verification, Matching Logic

1 INTRODUCTION

Modeling and verifying safety-critical systems [Kulik et al. 2022] can be a complex task that involves the use of diverse languages and abstraction levels. This complexity arises from the limitations of single languages, the complexity of systems, and the diversified requirement of different standards (e.g., ISO-15408[1][2] and IEC-61580 [Bell 2006]). For instance, the famous verified microkernel seL4 [Klein et al. 2014, 2010] proposes abstract specification in Isabelle, executable prototype in Haskell, and high-performance implementation in C.

*Corresponding author
[1]https://www.commoncriteriaportal.org/cc/index.cfm, accessed on July 1, 2024
[2]http://isotc.iso.org/, accessed on July 1, 2024

This work has been supported by the Natural Science Foundation of China under Grants No. U2341212 and No. 62132014.
Y. Zhao, P. Yao, and K. Ren also work at the State Key Laboratory of Blockchain and Data Security, Zhejiang University.
Authors' addresses: Jianhong Zhao, zhaojianhong96@gmail.com; Yongwang Zhao, zhaoyw@zju.edu.cn; Peisen Yao, pyaoaa@zju.edu.cn; Fanlang Zeng, flzeng@zju.edu.cn, School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China; Bohua Zhan, bzhan@ios.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Kui Ren, kuiren@zju.edu.cn, College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China.

ACM 1557-7392/2024/9-ART
https://doi.org/10.1145/3696000

Unfortunately, applying multiple models needs extra effort to synchronize models and verify their consistency. For example, seL4 introduces two steps to achieve a verified system. First, it transforms informal programs written in Haskell and C into the Isabelle/HOL formal framework. Second, it uses refinement verification to ensure the models accurately represent the same system. Despite these efforts, any changes to the Isabelle specification still require manual adjustments in Haskell and C. This repetitive and error-prone process requires repeated refinement verification to meet the stringent traceability requirements of safety-critical systems and international standards like ISO-15408. Moreover, this process becomes increasingly cumbersome, especially for developers not proficient in multiple languages, and is exacerbated by the growing number of models and iterations involved.

To tackle this problem, our vision is a formal bidirectional transformation (BX) framework. By inputting a formal consistency definition between m‘odels,z this framework produces synchronizers for *model synchronization* and formal proofs for rigorous *consistency verification* between the synchronized model pairs. Specifically, given model pair m and n, the synchronizer returns an n' (or m') that is consistent with m (or n) according to the predefined definition. Consistent parts in n (or m) remain unchanged, while inconsistent parts are updated in n' (or m') to achieve consistency. Models with incorrect syntax or semantics will not generate a consistent model, as the formal definition specifies consistency only between models with correct syntax and semantics. The synchronization process is governed by the predefined formal definition and validated by the generated proof. This approach allows developers to focus on their familiar models while ensuring the rigorous evaluation required for safety-critical systems. However, verified model synchronization in diverse languages and abstract levels via BX is extremely challenging, considering the following important criteria:

1. Expressiveness. Complicated synchronizations necessitate expressiveness to specify intricate consistency between heterogeneous models. To achieve this, BX frameworks should help developers focus on consistency without being burdened by extraneous concepts or implementation details. Moreover, there should be ample evidence demonstrating that developers can effectively construct diverse transformations using BX frameworks. However, the existing BX frameworks [Anjorin et al. 2020; Bettini and Efftinge 2016; Buchmann 2018; Buchmann et al. 2022; Cicchetti et al. 2011; Ko et al. 2016; Matsuda and Wang 2018; Weidmann et al. 2019] require provided manually parsers and printers, introducing extra implementation work and concept of abstract syntax trees. Additionally, few evidence shows that they can describe sophisticated model BX (e.g., UML and concurrent HCSP[3] BX) in practice.

2. Trustworthiness. Rigorous consistency verification for safety-critical systems demands a formal method with a small set of codes or other elements that must be trusted, known as the trust base. However, existing BX frameworks lack a formal proof system for consistency verification. Moreover, the existing formal frameworks such as Coq[4], Isabelle[5], Lean[6], and $\mathbb{K}$[7] offer rigorous formal consistency verification but lack solutions for verified model synchronization. First, they require a uniform BX model that can capture a reasonable BX within their theories. Second, they need synthesis algorithms to provide a minimized formal specification as a small trust base to generate a formal synchronizer. This synchronizer not only achieves model synchronization, as existing BX frameworks do, but also offers synchronization that adheres to a formal definition, supported by proofs.

Our key idea comes from the work of Chen et al. [2021a], which introduced the formal verification of the program execution via proof generation. By applying this approach, the $\mathbb{K}$ framework can automatically verify that the transformation complies with the established formal transformation definition. In addition to its verification

[3]HCSP (hybrid communicating sequential processes) [Chaochen et al. 1996; Liu et al. 2010] is an extension of CSP with ordinary differential equations, and it is widely used in verifying cyber-physical systems such as train control systems [Zou et al. 2015, 2013], cruise control system [Xu et al. 2022], and Mars lander [Zhan et al. 2021].
[4]https://coq.inria.fr/, accessed on July 1, 2024
[5]https://isabelle.in.tum.de/, accessed on July 1, 2024
[6]https://lean-lang.org/, accessed on July 1, 2024
[7]https://github.com/runtimeverification/k, accessed on July 1, 2024

capabilities, the $\mathbb{K}$ framework possesses a *language-oriented* nature, enabling the construction of intricate formal semantics intuitively. This feature has led to numerous programming languages, including C [Hathhorn et al. 2015], Java [Bogdanas and Roşu 2015], and JavaScript [Park et al. 2015], having their complete formal semantics defined in the $\mathbb{K}$ framework, rather than just partial semantics in other formal frameworks. Consequently, we can construct formal unidirectional transformation definitions within $\mathbb{K}$ to generate a verified transformer for model transformation while ensuring compliance with the formal definition.

Nevertheless, formal unidirectional transformation cannot deliver missing information recovery[8] and reverse transformation, as offered by bidirectional transformation for synchronization. Therefore, this paper introduces KBX, an extension to the $\mathbb{K}$ framework, enabling formal BX for verified model synchronization. To achieve this, we employ three steps: (1) Capture BX models using matching logic from $\mathbb{K}$ to facilitate the construction of formal BX $\mathbb{K}$ definitions. (2) Design formal BX definition synthesis algorithms to generate formal synchronizers from unidirectional transformation definitions. The BX definition incorporates both forward and backward transformation definitions to enable missing information recovery for the synchronized target. (3) Use formal synchronizers to maintain and verify the consistency of models simultaneously.

We initiate our evaluation of KBX by comparing its development efficiency and synchronization trustworthiness with current BX frameworks. This analysis highlights KBX's efficacy in constructing trustworthy BX programs. Next, we explore KBX's formal expressiveness and consistency verification reliability, demonstrating that we impose limited constraints on the expressiveness of $\mathbb{K}$ and maintain a small trust base during consistency verification. Lastly, we illustrate KBX's practicality in real-world scenarios by establishing the first formal BX between HCSP and UML. During the construction of this BX, KBX reduces the code size by 72%, compared with manual writing BX definitions.

To summarize, this paper makes the following main contributions:

- We present KBX, the BX framework for verified model synchronization, which generates formal synchronizers from formal unidirectional transformation definitions.
- We introduce a matching logic-based BX model to establish the relation between unidirectional definitions and BX definitions, as well as to specify the laws of BX definitions for synchronization. Furthermore, we present BX synthesis algorithms aimed at improving consistency verification trustworthiness and accelerating BX development. Based on the synthesis algorithms, we generate formal BX definitions from unidirectional ones in $\mathbb{K}$, producing formal synchronizers for simultaneous synchronization and consistency verification.
- We evaluate KBX's efficacy in synchronization and consistency verification. Additionally, we propose the first HCSP and UML BX between HCSP programs and PlantUML sequence diagrams for cyber-physical systems based on KBX, demonstrating its practical applicability in real-world scenarios. The implementation and evaluation of KBX can be accessed at https://github.com/Stevengre/kbx-public.

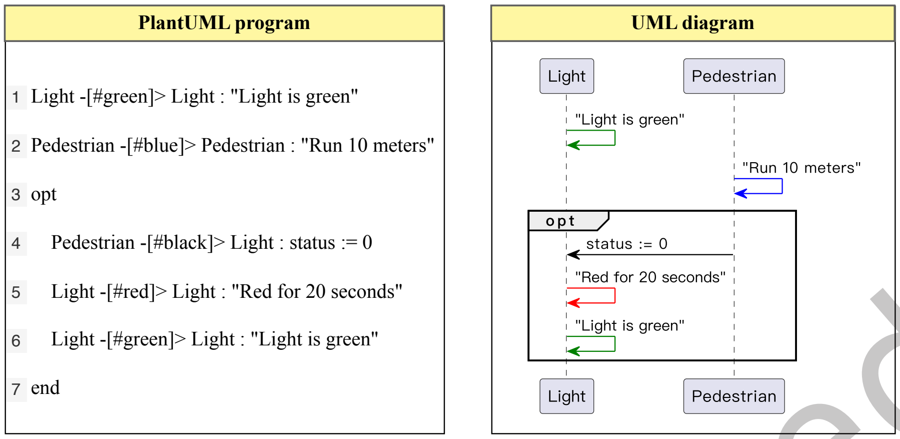
2 PRELIMINARIES

This section introduces the languages used in our motivating example (Section 3.1) and covers the foundational concepts of this work, including bidirectional transformation, matching logic, and the $\mathbb{K}$ framework.

2.1 PlantUML and HCSP: Languages Used in Motivating Example

During the development of safety-critical systems, engineers often employ different languages to balance intuitiveness and formality in system designs, providing the various models required by rigorous standards like

[8]For example, while a UML sequence diagram is a graph with colors, HCSP is not a graphical language and does not convey color information. This color information is missing in HCSP. However, during synchronization from HCSP to UML, it is necessary to recover the color information for UML.



```
Light -[#green]> Light : "Light is green"
Pedestrian -[#blue]> Pedestrian : "Run 10 meters"
opt
    Pedestrian -[#black]> Light : status := 0
    Light -[#red]> Light : "Red for 20 seconds"
    Light -[#green]> Light : "Light is green"
end
```

Fig. 1. A Sequence Diagram in PlantUML for Pedestrian Interaction with Traffic Signals.

ISO-15408 and IEC-61580. In our motivating example, we use two languages: PlantUML[9] and HCSP [Chaochen et al. 1996; Liu et al. 2010]. PlantUML is a text-based language for creating intuitive UML diagrams, while HCSP is a formal language for specifying, simulating, and verifying hybrid systems.

PlantUML is widely used in software engineering for its simplicity in creating UML diagrams, including class, sequence, and state diagrams. In our example, we focus on sequence diagrams to model the communication between system components.

Fig. 1 shows a PlantUML-generated sequence diagram illustrating a pedestrian-controlled traffic light system. The diagram includes two entities: a "Pedestrian" and a "Light". Initially, the "Light" is green, allowing cars to pass. After walking ten meters, the "Pedestrian" presses a button, changing the light to red for 20 seconds to allow safe crossing. The light then returns to green. This interaction is enclosed within an "opt" block, indicating that the initial button press triggers the subsequent actions.

$$
\begin{aligned}
\textit{sequence-diagram} &::= (\textit{sequence-statement}\backslash\mathbf{n}\ )^{*}\\
\textit{sequence-statement} &::= \textit{entity}\ \texttt{-}[\textit{color}]\texttt{>}\ \textit{entity}\ (\mathbf{:}\ \textit{string})?\\
&\mid\ \mathbf{'}\ \textit{anything}\ \mathbf{'}\\
&\mid\ \mathbf{loop}\ \textit{string sequence-diagram}\ \mathbf{end}\\
&\mid\ \mathbf{opt}\ (\textit{string})?\ \textit{sequence-diagram}\ \mathbf{end}\\
&\mid\ \mathbf{alt}\ \textit{string sequence-diagram}\ (\mathbf{else}\ \textit{sequence-diagram})?\ \mathbf{end}\\
&\mid\ \mathbf{group}\ \textit{string sequence-diagram}\ \mathbf{end}
\end{aligned}
$$

Fig. 2. Syntax for PlantUML's Sequence Diagram.

Fig. 2 illustrates the syntax for PlantUML's sequence diagrams, which include arrows connecting entities, comments, and control structures. Arrows can point to the same entity, indicating a self-initiated action, or to different entities, signifying an interaction between them. Comments provide descriptive text for the diagram.

[9]https://github.com/plantuml/plantuml, accessed on July 1, 2024

PlantUML's control structures—such as loop for repeating sequences, opt for optional sequences, alt for alternative sequences, and group for grouping sequences—enable the modeling of complex behaviors and the creation of blocks in diagrams.

While the UML diagrams are clear and intuitive, effectively describing system behaviors and component interactions, they fall short in precisely defining these behaviors. UML is a semi-formal language lacking the formal semantics necessary to specify, simulate, and verify system behaviors rigorously. To address this limitation, we introduce the HCSP language.

HCSP (Hybrid Communicating Sequential Processes) [Chaochen et al. 1996; Liu et al. 2010] exntends Hoare's Communicating Sequential Processes with differential equations, time constructs, interruptions, and more. It allows for the description of complex parallel and communicable components with both continuous and discrete behaviors. As a result, HCSP is widely used for modeling and verifying cyber-physical systems, such as train control systems [Zou et al. 2015, 2013], cruise control systems [Xu et al. 2022], and the Mars lander [Zhan et al. 2021].

```
Light ::=
  t := 0;
  log("Light is green");
  status := 1;
  (button ? status -->
    log("Red for 20 seconds");
    <t' = 1 & t <= 20>;
    status := 1;) ;
```

```
Pedestrian ::=
  s := 0;
  log("Run 10 meters");
  <s' = 1 & s <= 10>;
  button ! 0;
```

Fig. 3. HCSP Programs for Pedestrian Interaction with Traffic Signals.

Fig. 3 refines the pedestrian-controlled traffic light system using HCSP. The PlantUML entities are modeled as HCSP processes. The "Light" process includes a time variable "t" and a status variable "status" (0 for red, 1 for green). The "Pedestrian" process includes a distance variable "s". Informal information from PlantUML is retained in the log, while formal behaviors are specified using HCSP statements, such as assignments (:=) and continuous behaviors with differential equations and interruptions (e.g., <t' = 1 & t <= 20>). Communication between entities is modeled as individual actions on channels, such as receiving and sending status values through the button channel (button ? status and button ! 0, respectively).

The concrete syntax of HCSP, shown in Figure 4, consists of features described in [Liu et al. 2010] and includes additional syntactic sugars for convenience. HCSP programs are composed of a list of parallel HCSP processes whose behaviors are described by a sequence of HCSP statements. The HCSP statements include: (1) skip: a placeholder for skipping; (2) assignment; (3) wait: a convenience for continuous behavior, waiting for a certain time; (4) log: recording information; (5) communication statements: sending and receiving values from channels across processes; (6) if and if-then-else statements; (7) loop and loop-until statements; (8) interrupts caused by communication; (9) continuous behaviors described by differential equations; and (10) the combination of continuous behaviors and interrupts.

$$
\begin{aligned}
\textit{hcsp-program} &::= (\textit{hcsp-process}\ )^{*}\\
\textit{hcsp-process} &::= \textit{id}\ \mathbf{::=}\ (\textit{hcsp-statement}\ \mathbf{;})^{+}\\
\textit{hcsp-statement} &::= \mathbf{skip} \mid \textit{id} \mathrel{\mathbf{:=}} \textit{arithmetic-expression} \mid \mathbf{wait}\ \mathbf{(}\ \textit{int}\ \mathbf{)} \mid \mathbf{log}\ \mathbf{(}\ \textit{string}\ \mathbf{)}\\
&\quad \mid \textit{hcsp-io} \mid \textit{bool-expression}\ \mathbf{\text{->}}\ \textit{hcsp-statement}\\
&\quad \mid \mathbf{if}\ \textit{boolean-expression}\ \mathbf{then}\ (\textit{hcsp-statement}\ \mathbf{;})^{*}\ \mathbf{else}\ (\textit{hcsp-statement}\ \mathbf{;})^{*}\ \mathbf{endif}\\
&\quad \mid \mathbf{(}\ (\textit{hcsp-statement}\ \mathbf{;})^{*}\ \mathbf{)}\ \mathbf{**} \mid \mathbf{(}\ (\textit{hcsp-statement}\ \mathbf{;})^{*}\ \mathbf{)}\ \mathbf{\{}\ \textit{bool-expression}\ \mathbf{\}}\ \mathbf{**}\\
&\quad \mid \textit{interrupts} \mid \textit{continuous-behavior} \mid \textit{continuous-behavior}\ \mathbf{|>}\ \mathbf{(}\textit{interrupts}\ \mathbf{)}\\
\textit{hcsp-io} &::= \mathbf{id}\ \mathbf{?}\ \textit{id} \mid \mathbf{id}\ \mathbf{!}\ \textit{arithmetic-expression}\\
\textit{interrupts} &::= \textit{interrupt}\ (\mathbf{\$}\ \textit{interrupt})^{*}\\
\textit{interrupt} &::= \textit{csp-io} \mathbf{->} (\textit{csp-statement}\ \mathbf{;})^{*}\\
\textit{continuous-behavior} &::= \mathbf{<}\ \textit{differential-equations}\ \mathbf{\&}\ \textit{bool-expression}\ \mathbf{>}
\end{aligned}
$$

Fig. 4. Syntax for HCSP.

In summary, combining UML and HCSP is an effective approach for developing safety-critical cyber-physical systems. This choice is endorsed by our industrial partners, motivating this paper to address issues arising from the use of multiple languages and models, and to meet the high traceability requirements (consistency between models) mandated by standards like ISO-1540812 and IEC-61580. Our motivating example is detailed in Section 3.1.

2.2 Bidirectional Transformation

Bidirectional transformation (BX) enables the conversion of a system between two representations and maintains the synchronization of changes between them. For instance, it enables the synchronization of system designs between HCSP and plantUML by transforming one into the other while preserving information that is unique to each (i.e., missing information), such as the arrow colors in UML and the precise expressions in HCSP. A prominent theory elucidating bidirectional transformation is referred to as symmetric lenses [Hofmann et al. 2011], which permits missing information on both ends of such transformations. The *symmetric lens* is defined as follows.

Definition 1 (Symmetric Lens). Lens ℓ of model sets M, N (denoted as $\ell \in M \leftrightarrow N$) has three parts: a set of complements C, a distinguished element $missing \in C$, and two functions

$$putr\ :\ M \times C \to N \times C \qquad putl\ :\ N \times C \to M \times C$$

which satisfy the following inference rules as round-tripping laws:

$$\frac{putr(m,c) \quad = (n,c')}{putl(n,c') \quad = (m,c')} \tag{PutRL}$$

$$\frac{putl(n,c) \quad = (m,c')}{putr(m,c') \quad = (n,c')} \tag{PutLR}$$

The *missing* element in complements C represents missing information to be recovered for synchronization. For example, when synchronizing from HCSP to UML, we expect to update the modifications and retain the rest, including the missing information, in the UML model. The primitive unidirectional transformation ($M \to N$) can maintain the shared information but cannot recover the missing information, such as arrow colors in UML. In contrast, the forward transformation *putr* can recover the missing information (i.e., arrow colors in UML) from the complement C stored by the backward transformation *putl*, thus providing reasonable synchronization. The round-tripping laws formulate this synchronization rationale for forward and backward transformations.

2.3 $\mathbb{K}$ Framework

The $\mathbb{K}$ framework[10] is a language-agnostic framework for defining complex formal semantics and generating tools from them, such as printers, parsers, interpreters, and symbolic executors. This leads to various practical applications in programming languages, like C [Ellison and Grigore Roşu 2011; Hathhorn et al. 2015], Java [Bogdanas and Roşu 2015], and JavaScript [Park et al. 2015]. Its expressiveness, demonstrated by these applications, and its trustworthiness, established by [Chen et al. 2021a], motivate us to provide a formal bidirectional transformation approach in $\mathbb{K}$.

```
1 syntax HCSPStat ::= Id ":=" Expr | "log" "(" String ")"
2 syntax UMLStat ::= Id "-[" Color "]>" Id ":" String
3 configuration <m> $PGM:HCSP </m> <n> .K </n> <s> .K </s>
4 rule <m> log( A ); L := R ; HCSPs :HCSPStats => HCSPs </m>
5        <n> UMLs :UML => UMLs P -[#black]> P : A </n>
6        <s> P </s>
```

Fig. 5. A $\mathbb{K}$ snippet of HCSP to UML transformation for "log(A); L := R ;" to " P -[#black]-> P : A ".

Fig. 5 provides a code snippet in $\mathbb{K}$ to transform HCSP to PlantUML. Part of the syntax for HCSP and PlantUML is defined using the "**syntax**" keyword and BNF grammar in lines 1-2. Specifically, line 1 specifies "HCSPStat" for HCSP assignment and logging statements, while line 2 specifies "UMLStat" for a a colored arrow statement in PlantUML.

The "**configuration**" keyword in line 3 sets up the state's structure, initializing three cells: <m>, <n>, and <s>. The <m> cell is loaded with an environment variable "$PGM" (the HCSP program), the <n> cell stores the generated PlantUML, and the <s> cell stores the current process identifier. When using $\mathbb{K}$'s generated interpreter, the initial state should have the HCSP program in the <m> cell, with the <n> and <s> cells initially empty.

The "**rule**" keyword formalizes a rewriting rule in lines 4-6 for the HCSP to UML transformation. Because HCSP's concurrency semantics is quite complex, this example only considers one process within the HCSP program. Therefore, after other rules have processed the *hcsp-process*, we can obtain a concrete state that matches the left-hand side of the rule in lines 4-6, i.e., the abstract state on the left-hand side of the symbol "=>". Note that "<s> P </s>" in line 6 is a syntactic sugar for the "<s> P => P </s>".

For example, lines 3-4 in Fig. 3, "log("Light is green"); status :=1", can match the pattern on the left-hand side in line 4, because A , L , R , HCSPs are variables. Thus, " status :=1" matches with L := R . The symbol "=>" then performs the rewriting into the right-hand side in lines 4-6, resulting in a new state with the remaining HCSP statements (i.e., (*hcsp-process* **;**)*) in the <m> cell, the generated UML with the new arrow in the <n> cell, and the unchanged <s> cell.

From this example, we observe that $\mathbb{K}$ offers a concise and intuitive method for defining transformations between different languages. Consequently, it serves as the input for KBX. The trustworthiness of $\mathbb{K}$ is is grounded in matching logic, which will be discussed in the following subsection.

[10] https://github.com/runtimeverification/k, accessed on July 1, 2024

2.4 Matching Logic

Matching logic [Chen et al. 2021a,b; Chen and Rosu 2019] is the logical foundation of the $\mathbb{K}$ language framework, allowing for the specification and reasoning of programs. We use matching logic to capture the model and algorithm of KBX in Section 4 to ensure trustworthiness.

Matching logic formulas, called *patterns*, serve as unified structures for syntax and semantic specifications in $\mathbb{K}$. They are inductively defined based on the matching logic signature $\mathbb{\Sigma} = (EV, SV, \Sigma)$, where EV represents the set of element variables (denoted x, y, ...), SV denotes the set of set variables (denoted X, Y, ...), and Σ refers to the set of constant symbols (denoted σ, σ_1, ...).

Definition 2 (Matching logic syntax). The set PATTERN($\mathbb{\Sigma}$) of $\mathbb{\Sigma}$-patterns is inductively defined as follows:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\varphi_2 \mid \ \bot \ \mid \varphi_1 \rightarrow \varphi_2 \mid \exists x.\varphi \mid \mu X.\varphi$$

where φ has no negative occurrences of X in $\mu X.\varphi$.

Def.2 indicates that *patterns* are constructed with variables (x, X), constant symbols (σ), applications to apply the first argument to the second ($\varphi_1\varphi_2$), standard first-order logic (FOL) constructs ($\bot$, $\rightarrow$, $\exists$), and the least fixpoint construct (μ). Other basic notations are defined as follows:

$$\neg\varphi \equiv \varphi \rightarrow \bot \qquad \top \equiv \neg\bot \qquad \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \rightarrow \varphi_2$$
$$\forall x.\varphi \equiv \neg\exists x.\neg\varphi \qquad \nu X.\varphi \equiv \neg\mu X.\neg\varphi[\neg X/X] \qquad \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$$

Matching logic employs a *pattern matching semantics* [Chen et al. 2021b], interpreting a pattern φ as a set of elements that match it. For example, $\bot$ corresponds to $\emptyset$, and $\varphi_1 \wedge \varphi_2$ is matched by elements that match both φ_1 and φ_2. Similarly, " L := R " is matched by statements like " status := 1". When a match is found, we can apply a capture-free substitution of ψ (the matching element, e.g., "status" or "1") for x or X (the matched pattern, L and R) in φ. This substitution, denoted as $\varphi[\psi/x]$ or $\varphi[\psi/X]$, is defined in the usual way as follows:

$x[\psi/x] \equiv \psi$	$y[\psi/x] \equiv y$ if $y \neq x$
$\sigma[\psi/x] \equiv \sigma$	$(\varphi_1 \rightarrow \varphi_2)[\psi/x] \equiv \varphi[\psi/x] \rightarrow \varphi_2[\psi/x]$
$\bot[\psi/x] \equiv \bot$	$(\varphi_1\varphi_2)[\psi/x] \equiv (\varphi_1[\psi/x])(\varphi_2[\psi/x])$
$(\exists x.\varphi)[\psi/x] \equiv \exists x.\varphi$	$(\exists x.\varphi)[\psi/y] \equiv \exists z.\varphi[z/x][\psi/y]$ for fresh z
	$(\mu X.\varphi)[\psi/x] \equiv \mu Z.\varphi[Z/X][\psi/x]$ for fresh Z

The *matching logic proof system* [Chen et al. 2021a] defines a provability relation, written $\Gamma \vdash \varphi$, indicating that φ can be proved using the system, with patterns in a *matching logic theory* Γ as additional axioms. This system includes FOL rules, frame rules for application context reasoning, fixpoint rules for reasoning as in modal μ-calculus[Kozen 1983], and technical proof rules for completeness results [Chen and Roşu 2019].

Each $\mathbb{K}$ definition of a language L corresponds to a *matching logic theory* $\Gamma^L \subseteq$ PATTERN($\mathbb{\Sigma}$). This theory includes logical symbols representing the syntax of L and logical axioms specifying formal semantics. Transformation definitions, like Fig. 5, are matching logic theories and can be viewed as a special semantics of HCSP. $\mathbb{K}$ generates tools formally specified by matching logic formulas. The basic matching logic theories required by Γ^L include: equality as a basis, sorts for "**syntax**" and "**configuration**", and rewriting for "**rule**" and verification of program execution.

Equality refers to a (predicate) pattern $\varphi_1 = \varphi_2$ that holds (i.e., equals to $\top$) iff φ_1 equals to φ_2, and fails (i.e., equals to $\bot$) otherwise. To define equality, *definedness* represents a symbol $\lceil\_\rceil \in \Sigma$ with the axiom $\lceil\_\rceil x$, asserting that any element x is $defined$. Inductively, the predicate pattern $\lceil\_\rceil\varphi$ states that φ is *defined* (or φ is not $\bot$), i.e., φ is matched by at least one element. Various critical mathematical constructs are defined based on *definedness*, including *totality* $\lfloor\varphi\rfloor \equiv \neg\lceil\neg\varphi\rceil$, *equality* $\varphi_1 = \varphi_2 \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor$, *membership* $x \in \varphi \equiv \lceil x \wedge \varphi\rceil$, and *inclusion* $\varphi_1 \subseteq \varphi_2 \equiv \lfloor\varphi_1 \rightarrow \varphi_2\rfloor$.

Sort s is specified by the *inhabitant pattern* $[\![s]\!]$ (abbreviated for the application $[\![\_]\!]s$) with a symbol $[\![\_]\!] \in \Sigma$ to capture sorted $\mathbb{K}$. For example, in Fig. 5, line 1 defines the sort "HCSPStat" with the inhabitant pattern $[\![HCSPStat]\!]$ to denote the set of all possible HCSP statements. Similarly, the "**configuration**" keyword defines a sort for the computation state.

Rewriting employs a *one-path next* symbol $\bullet \in \Sigma$ to capture the transition system (or formal semantics) over computation states defined by rewrite rules in $\mathbb{K}$. This symbol denotes that for any state γ, $\bullet\gamma$ is matched by all state that can go to γ in one step. In other words, γ is reached on *one-path* in the *next* state. Using the symbol $\bullet$, we can define the rewriting as follows:

Definition 3. Rewriting (i.e., program execution) is the reflexive and transitive closure of one-path next, which can be defined as follows:

$$\diamond\varphi \equiv \mu X.\varphi \vee \bullet X \text{ // Eventually; } \varphi \vee \bullet\varphi \vee \bullet\bullet\varphi \vee \dots$$
$$\varphi_1 \Rightarrow \varphi_2 \equiv \varphi_1 \rightarrow \diamond\varphi_2 \text{ // Rewriting}$$

Rewriting can also be referred to as *program execution* in $\mathbb{K}$ because it describes the execution trace of a program from the initial state φ_{init} to the final state φ_{final}. Within this trace, patterns $\varphi_{init}, \dots, \varphi_{final}$ serve as execution *snapshots*. For each step from φ_i to φ_{i+1} (e.g., transforming HCSP in Fig.3 to PlantUML in Fig.1), the formal semantics Γ^L provides a rewriting axiom through a rewrite rule $\varphi_{lhs} \Rightarrow \varphi_{rhs}$ (e.g., lines 4-6 in Fig. 5) and the corresponding substitution (e.g., [status/ L] and [1/ R] for " L := R ").

For trustworthiness, [Chen et al. 2021a] define the correctness of the execution from φ_{init} to φ_{final}, witnessed by a formal proof: $\Gamma^L \vdash \varphi_{init} \Rightarrow \varphi_{final}$. This ensures that the final state is reached from the initial state according to matching logic theory. They also propose a method to generate such proofs for given execution traces, which can be verified by a trustworthy proof checker. Specifically, they selected Metamath, a third-party proof checking tool, known for its simplicity, speed, and reliability, to formalize matching logic and encode its proofs. Metamath's proof checkers, typically only a few hundred lines of code, can verify thousands of theorems per second, making it an ideal tool for verifying execution trace correctness. This method, therefore, ensures execution correctness through proof generation.

In the context of transformation, the transformed PlantUML (i.e., φ_{final}) is consistent with the HCSP program (i.e., φ_{init}) according to the formal consistency definition (Γ^L). Thus, we employ $\mathbb{K}$ generated interpreters to provide verified synchronization, ensuring that the synchronized models are consistent under the formal consistency definition. Additionally, the generated proof can serve as evidence for evaluating traceability (i.e., consistency between different models) required by standards like ISO-1540812 and IEC-61580.

3 RESEARCH OVERVIEW

In this section, we first clarify the motivation of our work via an industrial example from high-speed maglev train verification. We then present an overview of KBX and state the challenges of verified model synchronization.

3.1 Motivating Example

This work is motivated by an industrial practice of verifying high-speed maglev trains, which are heterogeneous safety-critical systems with frequent and complex communications. To ensure the intuitive yet rigorous design of such systems, we collaborate with our industrial partners and utilize two distinct models: (1) an abstract model in UML that is rough but intuitive for team communication; and (2) a refined model in HCSP, a formal language for hybrid system simulation and verification.

For instance, in Fig 6, the UML model visualizes how pedestrian-controlled traffic lights work. In this model, arrows pointing to oneself represent actions, like "*Run* 10 *meters*", while arrows pointing to others signify interactions initiated by an entity, as seen in "$status := 0$". The "*opt*" block indicates that the first message can

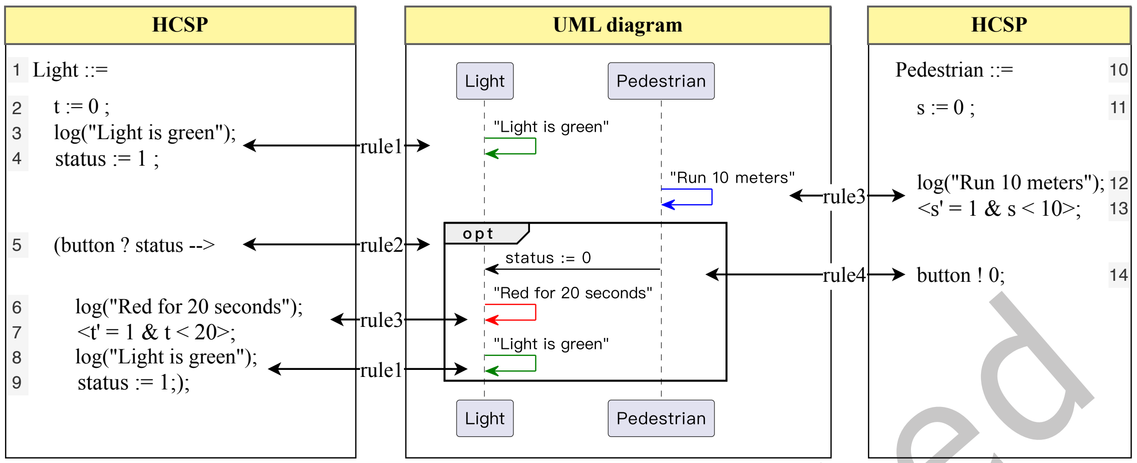


Fig. 6. Pedestrian Interaction with Traffic Signals: A depiction of a pedestrian crossing the road with a traffic light initially displaying red, followed by a 10-meter walk, button activation, a 20-second green signal, and subsequent return to red.

trigger operations in it. In contrast, the HCSP model refines the UML model by applying specific rules (e.g., “*rule*1” converts the message content of UML “*Light is green*” to the log of HCSP, and introduces an expression to refine this behavior “$status := 1$”). This refined model introduces elements like assignments, e.g., “$status := 1$”, channels, e.g., “$button\ ?\ status$”, and ordinary differential equations, e.g., “$< s' = 1\ \&\ s < 10 >$”. By exiling details from abstract models to refined models, the combined use of HCSP and UML improves both clarity and precision in the design process.

However, using multiple models suffers from trustworthiness issues. As an example, the HCSP model may not synchronize with the UML model according to rule4. After the pedestrian presses the button, the light should switch to red, ensuring the pedestrian crosses the road. However, due to the lack of “$button\ !\ 0$”, the light erroneously remains green, posing a substantial risk of traffic accidents. As intricate and safety-critical systems, Maglev trains could lead to even graver consequences if consistency is compromised. Hence, it becomes imperative to synchronize these models and rigorously verify their consistency.

Model Synchronization. Our first task is to maintain consistency by synchronizing the models after any modification. This involves shared information updates and missing information recovery. For instance, if we adjust the pedestrian distance traveled in HCSP from 10 to 5 (lines 12-13 of Fig 6), we need to update the message content (shared information) to “*Run* 5 *meters*” and recover the message color (missing information) to be blue according to “*rule*5” in UML. Manually updating all the modifications in shared information is laborious, while developing separate unidirectional transformers with mutually consistent behavior for synchronization can be difficult and error-prone. Using BX frameworks enables the efficient development of bidirectional transformers that satisfy synchronization rationale (e.g., round-tripping laws) for accurate update and recovery.

Consistency Verification. Our second task is to prove the semantic consistency between synchronized models. This is typically achieved through refinement verification, which ensures that the state sets of synchronized models adhere to a predefined state relation before and after symbolic simulation, based on the models’ formal semantics. This process is often carried out using interactive theorem provers and requires numerous manually constructed proofs to ensure trustworthiness. Additionally, any modifications to the models necessitate repeating the verification process, resulting in substantial verification efforts. For example, proving semantic consistency between HCSP and UML through refinement verification involves the following steps:

(1) Construct formal semantics for HCSP and UML within a formal framework.
(2) Develop translators to convert HCSP and UML models into formal representations.
(3) Define formal specifications for the consistency between HCSP and UML, i.e., the relation between state models of HCSP and UML semantics.
(4) Construct proofs to verify that the synchronized models adhere to the formal specifications:
    - Simulate the HCSP program from any initial state S_{hcsp} to its final state S'_{hcsp} via the formal semanitcs of HCSP.
    - Simulate the UML model from any initial state S_{uml} to its final state S'_{uml} via the formal semantics of UML.
    - Prove that the initial states S_{hcsp} and S_{uml}, as well as the final states S'_{hcsp} and S'_{uml}, conform to the predefined state relation.
(5) Manually reconstruct proofs after each model modification.

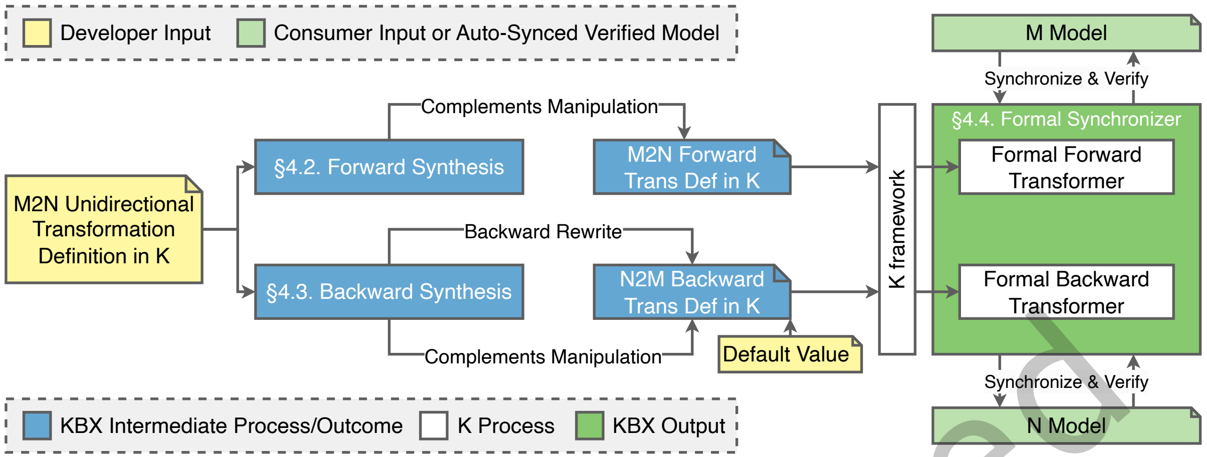
In summary, employing multiple modeling languages allows us to take the unique strengths of each language, but it also introduces complexities that can hamper the development and verification process, leading to inefficiencies and the risk of errors.

3.2 KBX Overview

To simplify the development and verification of multiple models, our vision is to design a formal bidirectional transformation (BX) framework. First, by automating the synchronization process, BX eliminates the laborious manual language conversions, such as those between PlantUML and HCSP. Second, formal BX takes a step further by addressing the need for *consistency verification*. This is accomplished by employing formal specifications to establish rigorous consistency and utilizing a proof system to verify this consistency. As a result, the framework can (1) automatically synchronize HCSP and UML models, and (2) formally prove the consistency.

However, the existing BX frameworks face limitations in both synchronization and verification.

- First, existing BX frameworks fall short in expressiveness and trustworthiness for *synchronization*. Specifically, they lack a focus on being *language-oriented* and *formal*. To illustrate the issue of expressiveness, consider the synchronization of HCSP and UML models. Using current BX frameworks, users not only have to implement BX programs but are also required to develop parsers and printers for both HCSP and UML. These parsers enable the BX program to manipulate the models, while the printers produce HCSP and UML representations instead of abstract syntax trees (AST). A more expressive BX framework should eliminate the need for users to develop parsers and printers. Additionally, in terms of trustworthiness, these frameworks often rely on programming languages (e.g., C# [Hinkel and Burger 2019]) , diagram languages (e.g., EVL+Strace [Samimi-Dehkordi et al. 2018]), or Meta-Object Facility-based languages (e.g, Vitruvius [Klare et al. 2021]) to describe model relations, rather than formal languages like $\mathbb{K}$, Coq, and HOL. This can result in undefined behavior of the generated synchronizer.
- Second, current BX frameworks lack formal languages and proof systems, making *consistency verification* difficult and unreliable. Without formal languages, setting verification goals is impossible, and without proof systems, proving these goals is unfeasible. As Stevens [2008] noted, "most work on verification or validation of model transformations is not about ensuring the transformation conforms to its specification and user requirements, but about ensuring the transformation is sane." This remains the case today. Approaches discussed by [Hidaka et al. 2016; Hildebrandt et al. 2013] only address algorithm correctness. Even Ko et al. [2016] used an interactive theorem prover to verify synthesis algorithm correctness, unlike earlier studies such as Hermann et al. [2011] which provided manual proofs. This lack of formal verification



Developer Input
Consumer Input or Auto-Synced Verified Model
M Model
Synchronize & Verify
Complements Manipulation
§4.2. Forward Synthesis
M2N Forward Trans Def in K
M2N Unidirectional Transformation Definition in K
Backward Rewrite
§4.3. Backward Synthesis
N2M Backward Trans Def in K
Complements Manipulation
Default Value
K framework
§4.4. Formal Synchronizer
Formal Forward Transformer
Formal Backward Transformer
Synchronize & Verify
N Model
KBX Intermediate Process/Outcome
K Process
KBX Output

Fig. 7. The Architecture of KBX Formal BX Framework.

for synchronization processes results in unpredictable outcomes for BX programs, as shown by [Anjorin et al. 2020].
Given the complexity and errors already identified in BX programs, relying on them as a trust base is unreliable. Additionally, the complexity of these programs and their implementation languages makes formal verification practically impossible. Thus, laborious consistency verification appears inevitable, as discussed in Section 3.1. However, even if such efforts are tolerated, the newly introduced translators from the informal to the formal world increase the trust base, which cannot be verified. This conflicts with the requirements for safety-critical system development, which aims to minimize the trust base.

Approach. This paper introduces KBX, a novel formal BX framework for simultaneous synchronization and consistency verification, as shown in Fig 7. In KBX, developers use unidirectional transformation definitions to generate formal synchronizers, automating model synchronization and consistency verification for consumers. This integration of formal consistency verification not only ensures trustworthiness but also aligns it with high evaluation assurance levels of international safety and security standards, e.g., IEC-61580 [Bell 2006], DO-178C [Jacklin 2012], and ISO-15408[11], making it particularly suitable for safety-critical systems. To elucidate KBX's trustworthiness and expressiveness, the following delves into KBX workflow in Fig. 7.

Expressiveness. KBX simplifies the development of formal synchronizers by using a unidirectional transformation definition in $\mathbb{K}$ to establish formal consistency between synchronization targets. For instance, developers only need to provide rewrite rules to transform HCSP to PlantUML, defining the formal axioms of their consistency. As illustrated in Fig. 3, the consistency between "log(A); L := R ;" and " P -[#black]-> P : A " is formally defined for *rule*1 in Fig. 6.

KBX uses these formal consistency definitions to automatically generate formal forward and backward transformation definitions, preserving the specified consistency. During the formal synthesis stage (Section 4.2), KBX enriches each rule with additional manipulations to produce a forward transformation definition that replicates the original transformation while capturing and recovering any missing information. Conversely, the backward synthesis process (Section 4.3) produces a backward transformation definition in harmony with the

[11]https://www.commoncriteriaportal.org/cc/index.cfm, accessed on July 1, 2024

forward transformation, ensuring compliance with round-tripping laws. This compliance is achieved through the backward rewrite that we proposed in matching logic.

After generating the backward transformation definition, developers should specify default values within it. Although KBX provides a symmetric lens, the original consistency definition only supplies default values missing in the forward transformation. For example, the default arrow color in PlantUML is "#black", and default values for HCSP assignments, such as `L` and `R`, are not specified. For these values, KBX generates placeholders like "?KbxGenTodo0" for `L` and "?KbxGenTodo1" for `R` in the backward transformation definition.

Once the default values are filled in, KBX leverages the $\mathbb{K}$ framework to automatically generate verifiable and certifiable transformers from these definitions, forming a formal synchronizer for synchronization and consistency verification. Note that, the generated backward transformation definition is valid with generated placeholders and can be used to create symbolic synchronized models, such as HCSP programs with symbolic values. For better efficiency, it is preferable to replace these placeholders with actual default values to obtain a concrete synchronized model using the $\mathbb{K}$-generated concrete interpreter.

Consequently, KBX's *expressiveness* for developers is manifested in three key aspects: conciseness, intuitiveness, and capability. KBX's synthesis processes eliminate the need for developers to manage the complexities of synchronization rationale and reverse transformation definitions, enhancing conciseness. Its intuitive nature, as shown in lines 4-5 of Fig. 5, enables developers to use target syntaxes like HCSP and UML for formalizing consistency, avoiding the challenges of handling dual ASTs and reducing the trust base. Finally, KBX's capability is grounded in the established efficacy of the $\mathbb{K}$ framework, which has been successfully applied in the formal semantics of languages including C, Java, and JavaScript.

Trustworthiness. Beyond its expressiveness, KBX establishes trust by proving the consistency between synchronized models through the $\mathbb{K}$ framework's proof generation capability. The generated proof is verified by a proof checker to ensure that the execution by the $\mathbb{K}$-generated interpreter conforms to its formal semantics. In synchronization, this verification ensures that each synchronization process adheres to predefined formal consistency definitions. Consequently, synchronized model pairs are validated to be consistent under these formal definitions.

Conversely, if the proof does not pass the proof checker's examination, the synchronized model pairs are deemed inconsistent. There are several reasons for this: (1) If the input syntax is incorrect, the verification will fail because the incorrect syntax is not defined in the consistency definition. For instance, only the correct syntax of HCSP and PlantUML is defined as a valid matching logic pattern in the consistency definition. Therefore, in forward transformation, only syntactically correct HCSP can be synchronized into syntactically correct PlantUML. (2) If the semantics (both static and dynamic) are incorrect, there will be no rewrite rule as matching logic axioms to transition the input model to the output model. For example, if an HCSP program contains only a send message, the transformation process will deadlock, preventing further transformation and thus failing to produce a synchronized PlantUML model. (3) Even if an erroneous model is synchronized, it will be detected because the proof checker cannot validate the proof.

Our approach reduces the synchronization trust base from complex BX programs to intuitive, easily-reviewable formal consistency definitions and proof checkers. This method relies on a formal framework for consistency verification that is comparable to semantics-based refinement verification. With reliable formal semantics, these definitions can be verified, further reducing the trust base. When targets lack unambiguous semantics, the formal definitions serve as the minimal trust base. Notably, our verification method does not depend on the correctness of SMT solvers. Instead, we use the meticulously selected Metamath proof checker, known for its simplicity and reliability, typically comprising only a few hundred lines of code and having multiple open-source implementations. This enhances the trustworthiness of our consistency verification, providing rigorous certification and simplifying the evaluation process, which is crucial for safety-critical systems.

Summary. KBX offers an expressive framework for developers to construct formal synchronizers, coupled with a trustworthy method for consumers to verify model consistency. The following subsection will present a problem formulation for achieving simultaneous synchronization and consistency verification. It will outline the technical challenges encountered and provide a roadmap for addressing these.

3.3 Problem Statement

In this study, we tackle the FSS (Formal Synchronizer Synthesis) problem, focusing on the consistency verification and synchronization between model sets M and N, exemplified by programs in HCSP and models in PlantUML. A key element to this problem is the complements C, indicative of information gaps inherent between M and N — crucial for differentiating BX from normal transformation.

FSS Problem Statement

Given a forward transformer $ux : M \rightarrow N$ compliant with formal definition Γ^{ux}, we aim to synthesize a formal synchronizer $\ell \in M \leftrightarrow N$ that comprises a forward transformer $putr : M \times C \rightarrow N \times C$, a backward transformer $putl : N \times C \rightarrow M \times C$, and a strategy for managing complements C. The transformers $putr$ and $putl$ should satisfy these conditions: (1) Compliance with their respective formal definitions Γ^{putr} and Γ^{putl}, verified by a proof checker. (2) The output model $n \in N$ from $putr$ mirrors that of ux for identical input models, given an empty input complement. (3) Adherence to the round-tripping laws for reasonable synchronization.

Following the problem statement, we distill the key to our work: (1) the formalization and verifiability of each definition for BX, and (2) the automatic provability of every executable tool derived from these definitions. We achieve this by synthesizing formal definitions within the $\mathbb{K}$ framework, harnessing its capabilities for generating interpreters, verification tools and proofs. The input is a language-agnostic $\mathbb{K}$ definition, covering syntax for both source and target transformations, state declarations, and transformation rules. This allows us to utilize existing $\mathbb{K}$ semantics (e.g., Java, C, AADL) to validate related BX correctness and reuse portions of syntax and semantic rules, including static semantics, for BX construction. However, synthesizing formal BX definitions (i.e. forward and backward transformation definitions Γ^{putr} and Γ^{putl}) and crafting a formal BX framework is far from trivial.

Roadmap. Our work addresses the following key challenges in tackling the FSS problem:

Matching Logic-based BX Model (Section 4.1). The $\mathbb{K}$ framework, initially tailored for programming language semantics and verification, lacks direct support for BX formalization. Thus, our first step is to model the FSS problem using matching logic to apply $\mathbb{K}$'s advanced formal methods, thereby satisfying the first FSS condition. We strive to formalize a versatile model capable of handling multifaceted and multilingual BX scenarios beyond just HCSP and PlantUML, ensuring that our approach retains the expressive power of $\mathbb{K}$ without imposing undue restrictions.

Forward Transformation Synthesis (Section 4.2). Building on our basis model, we seek to synthesize the forward transformer $putr$ addressing missing information recovery (e.g., recapturing the omitted PlantUML message color in HCSP). Altering the existing transformer ux directly would contravene its formal definition Γ^{ux}, thus affecting its verifiability. We propose deriving a human-readable $\mathbb{K}$ forward definition Γ^{putr} from Γ^{ux}. This endeavor introduces two primary challenges: (1) Developing a method for manipulating complements C within Γ^{putr} that aids in information recovery while preserving $\mathbb{K}$'s language-agnostic nature to facilitate verified synchronization across various models. (2) Ensuring that the inclusion of complements manipulation in $putr$ maintains its output equivalence to ux under identical inputs, satisfying the second FSS condition.

Backward Transformation Synthesis (Section 4.3). Essential to our formal synchronizer, the backward transformer $putl$ confronts challenges in handling language-agnostic models and satisfying the round-tripping laws. We respond to these challenges by introducing the "backward rewrite" within matching logic, harnessing matching logic's capacity to transcend language specifics. This approach is crucial for synthesizing Γ^{putl} and ensuring $putl$ satisfies the round-tripping laws.

Formal Synchronizer (Section 4.4). Although the formal transformers $putr$ and $putl$ are effectively derived from Γ^{putr} and Γ^{putl}, the complexity of the $\mathbb{K}$ framework and its toolchains presents challenges in practical application. To counter this, we introduce a formal synchronizer strategy that facilitates both synchronization and consistency verification processes. This synchronizer not only streamlines the use of $putr$ and $putl$ as a symmetric lens but also generates consistency proofs for synchronized models (e.g., verifying if HCSP and UML models in Fig. 6 represent the same system). These proofs, verifiable by a proof checker, offer tangible evidence of consistency, elevating the assurance level of the evaluation for safety-critical systems.

Remark. Introducing BX to a formal framework presents several advantages:

Trustworthy Synchronization with a Small Trust Base. (1) *Intrinsic Verifiability.* Definitions related to synchronization are readable, amenable and verifiable to customization and review. This diminishes the necessity for developing and trusting translations to other languages with a proof system. (2) *Direct Correctness Verification.* This allows developers to define and verify critical transformation properties directly. Where operational semantics of synchronization targets are trustworthy, they can serve as an ideal basis for demonstrating synchronization correctness. (3) *Trustworthy Execution.* The reliability of the synchronization process is ensured by generating proofs that are subsequently validated using a trusted proof checker.

Automatic Verification Comparable to Refinement Verification. As discussed in Section 4.4, we elucidate how consistency verification through KBX aligns with, and is comparable to, refinement verification. This section also outlines methodologies for proving their compliance.

4 APPROACH

This section begins by formalizing the FSS problem using matching logic (Section 4.1), laying the logical basis for the paper. We then introduce our solution, including three steps: the synthesis of forward (Section 4.2) and backward (Section 4.3) transformations, and their use in simultaneous consistency verification and synchronization (Section 4.4). Additionally, Section 4.4 explores the meaning of consistency verification results in KBX.

4.1 Matching Logic-based Bidirectional Transformation Model

This section presents the KBX model, the logical foundation of the KBX framework, to formulate the FSS problem in the $\mathbb{K}$ framework. This model includes two matching logic theories Γ^{putr} and Γ^{putl} by reforming the unidirectional one Γ^{ux}. Our synthesis uses this model to generate the formal BX definitions from the unidirectional ones in $\mathbb{K}$.

KBX Input: Unidirectional Transformation Definition Γ^{ux}. To provide an expressive and intuitive way to construct BX, the KBX model introduces a unidirectional definition Γ^{ux} to capture the consistency definition within $\mathbb{K}$. This definition defines the transformations $ux : M \to N$ via matching logic rewriting $m\ \sigma_n\ \sigma_s \Rightarrow \sigma_m\ n\ s$. These transformations begin with an input model m, an empty output model σ_n, and an initial transformation state σ_s. By substituting the rewrite rules in Γ^{ux}, these transformations result in an empty input model σ_m, an output model n, and a final transformation state s. The following definition describes Γ^{ux}'s function and enables users to define consistency using $\mathbb{K}$ rewrite rules without limitations.

Definition 4. Given the sorts M, N, S and constants $\sigma_m : M, \sigma_n : N, \sigma_s : S$, unidirectional transformation definition Γ^{ux} results in the following rewriting:

$$m : M, n : N, s : S.\ \Gamma^{ux} \vdash m\ \sigma_n\ \sigma_s \Rightarrow \sigma_m\ n\ s$$

In this context, the symbols M, N, and S denote the syntax sorts for the input model, output model, and transformation helper, respectively. These sorts, such as the syntax of HCSP and plantUML, are defined within the Γ^{ux} definition. Additionally, the definition includes a set of rewriting rules that facilitate the transformation process through the rewriting theory. The definition configuration declares the initial state and the structure of the states in the rewriting rules.

This formalization strikes a balance between specificity and generality. It is detailed enough to define a transformation, yet flexible enough to accommodate diverse languages and semantics. This is achieved without imposing limitations on the sorts M, N (which define the syntax of transformation targets) and the rewrite rules in Γ^{ux} (which determine the semantics of consistency). For instance, in Fig. 5, the syntax like "HCSPStat" is captured by M, N. The "**configuration**" (line 3) specifies the initial transformation state as $m\ \sigma_n\ \sigma_s$, and the "**rule**" (lines 4-6) illustrates one of the Γ^{ux} rules that govern the transformation process. Developers can adapt this model to other transformations using "**syntax**" to define M, N, S, "**configuration**" to set the transformation state, and "**rule**" to delineate transformation semantics.

KBX Output: Bidirectional Transformation Definitions Γ^{putr} and Γ^{putl}. We present Def. 5 and Def. 6 to describe the bidirectional transformation definitions in matching logic.

First, we clarify the concept of forward definition Γ^{putr} by establishing its relation with Γ^{ux}. The following definition guarantees that when applying the transformations defined by Γ^{ux} and Γ^{putr} to an input model m, it will result in an identical transformed model n. Consequently, the forward definition Γ^{putr} shares the same consistency definition as the unidirectional definition Γ^{ux}.

Definition 5. Given the sorts M, N, S, C and constants $\sigma_m : M, \sigma_n : N, \sigma_s : S$, the forward definition Γ^{putr} mirrors the behavior of the unidirectional definition Γ^{ux}: $\forall m : M, n : N, s : S. \exists c : C, c' : C$.

$$\frac{\Gamma^{ux} \vdash \quad m\ \sigma_n\ \sigma_s \Rightarrow \sigma_m\ n\ s}{\Gamma^{putr} \vdash \quad m\ \sigma_n\ c\ \sigma_s \Rightarrow \sigma_m\ n\ c'\ s}$$

Second, we obtain the correct bidirectional definitions by refactoring the round-tripping laws from Definition 1 with matching logic results in Definition 6. The definition Γ^{putr} achieves the same directional transformation as Γ^{ux}, while Γ^{putl} accomplishes the reverse transformation. Unlike the unidirectional definition Γ^{ux}, the bidirectional definitions Γ^{putr} and Γ^{putl} incorporate the sort C and its corresponding configuration to retain information that might be lost during the transformation. Furthermore, according to PuxRL, it is ensured that Γ^{putr} accurately preserves information, and Γ^{putl} effectively restores information. Similarly, PuxLR guarantees that Γ^{putl} correctly preserves information, and Γ^{putr} aptly recovers information.

Definition 6. Given the sorts M, N, S, C and constants $\sigma_m : M, \sigma_n : N, \sigma_s : S$, bidirectional transformation definitions Γ^{putr}, Γ^{putl} satisfy the following laws: $\forall m : M, n : N, c : C, c' : C, s : S$.

$$\frac{\Gamma^{putr} \vdash \quad m\ \sigma_n\ c\ \sigma_s \Rightarrow \sigma_m\ n\ c'\ s}{\Gamma^{putl} \vdash \quad \sigma_m\ n\ c'\ s \Rightarrow m\ \sigma_n\ c'\ \sigma_s} \tag{PuxRL}$$

$$\frac{\Gamma^{putl} \vdash \quad \sigma_m\ n\ c\ \sigma_s \Rightarrow m\ \sigma_n\ c'\ s}{\Gamma^{putr} \vdash \quad m\ \sigma_n\ c'\ s \Rightarrow \sigma_m\ n\ c'\ \sigma_s} \tag{PuxLR}$$

In conclusion, we introduce the input definition Γ^{ux} and the output definitions $\Gamma^{putr}, \Gamma^{putl}$ for KBX, aligning it with the scope of matching logic. Given that each matching logic definition corresponds to a $\mathbb{K}$ definition, developers can utilize the $\mathbb{K}$ framework to conveniently define consistency and implement the unidirectional transformation. This involves using the ***syntax*** keyword to define sorts M, N, and S through EBNF-like grammar; the ***configuration*** keyword to identify constants σ_m, σ_n, and σ_s using XML-like grammar; the ***rule*** keyword to establish rewrite/semantic rules within Γ^{ux}. By providing these $\mathbb{K}$ definitions, the $\mathbb{K}$ system automatically generates transformation and verification tools for automatic verification and synchronization.

Nevertheless, composing bidirectional transformation definitions Γ^{putr}, Γ^{putl} poses challenges. Firstly, designing and implementing complements is time-consuming and error-prone. Secondly, managing numerous rewriting rules presents difficulties in ensuring accurate information storage and recovery. These factors contribute to a substantial workload and reduced reliability. Hence, we propose the subsequent synthesis to address these issues.

4.2 Forward Transformation Synthesis

This section delves into the design of a complement structure and the synthesis workflow for the forward transformation, denoted as *putr*, within the KBX framework.

Structure of the Complements Holder. Complements C refers to the missing information to be recovered during the forward (*putr*) and backward (*putl*) transformation. In KBX, we manipulate this missing information in both directions of a BX by using a map that links common information to distinct information on either side of a rewrite rule, represented as *common* $\mapsto$ *missing*. To illustrate this in the language-agnostic framework $\mathbb{K}$, consider the case of unidirectional transformation definition Γ^{ux}. This definition, as exemplified by lines 4-6 in Fig. 5, employs a set of rewrite rules to define transformation semantics. For each rule within $\mathbb{K}$ definitions, various transformation-related information types are identifiable: (1) Variables/Tokens (e.g., A , L , “#black”), (2) Syntax productions like “log (_)” and “_:=_” and , (3) State productions, which are syntactic sugars for generating “Cell” patterns (e.g. “<m> _ </m>”), (4) Side conditions that establish preconditions and postconditions, and (5) Rule attributes (e.g. “ priority (_)”, which determine the rule adoption order).

Discerning Missing Information. We identify the *missing* information by examining the asymmetry around the rewrite symbol “=>” within the $\mathbb{K}$ definition Γ. This process involves dissecting Γ into individual assessments of each rewrite rule. Such segmentation is reasonable because the transformation process comprises distinct *snapshots*, each corresponding to the application of a rewrite rule (as detailed in Section 2.4). Consequently, any information not present post-application of a rule is deemed *missing* after the transformation. Our analysis across different information types leads to the following conclusions[12]:

- Variables/Tokens are identified as *missing* if they are not present on both sides of the rewrite symbol “=>”. For instance, L and R in the *ux* transformation targets are considered *missing* as their values cannot be deduced from the right side of the rewrite rule.
- Syntax productions are considered *missing* when there is an overlap in matched pattern sets on the right of two rewrite rules. This overlap impedes distinguishing syntax productions on the left side from the other. We presuppose such overlaps do not occur on the left side of Γ^{ux}'s rewrite rules to maintain deterministic transformation over symbolic results.
- Other information types like state productions, rule attributes, and side conditions are not categorized as *missing*. State productions and rule attributes remain constant through the rewriting process, and while side conditions influence rule application, they do not modify the state. Thus, they do not contribute to the *missing* information during the transformation.

Designing Complements Holder. Identifying *missing* information is a critical initial step, but it is insufficient for forming a practical structure for information recovery. To address this, we use a “Map” in the $\mathbb{K}$ framework, structured as *common* $\mapsto$ *missing*, ideal for data manipulation.

The *common* element represents shared information across both sides of the rewrite rules, facilitating consistent handling of the corresponding *missing* information. However, defining what constitutes *common* is crucial. Overburdening the “Map” with excessive data leads to impractical key sizes. Therefore, we selectively identify *common* elements:

- *Variables/Tokens*. Included in *common* when appearing on both sides across different states, exemplified by A and P in Fig. 5. Conversely, variables/tokens common within the same state are excluded. For instance, HCSPs and UMLs , which merely serve as the context of rewrite rules, are excluded. Including

[12]Note that this analysis takes into account both directions of synchronization

such context-specific variables in *common* would bloat *common* and reduce its functionality, especially when slight changes in context render the complements ineffective for information recovery. For example, in Fig. 6, if we retain HCSPs and UMLs , then when adopting rule1 to transform lines 3-4 of HCSP into the corresponding PlantUML, all code after line 4 in HCSP and the already transformed PlantUML would need to be stored as *common* in the complements holder. This not only increases size but also means that "status" and "1" can only be recovered if the modifications in PlantUML do not affect the code after line 4 in HCSP. This is unreasonable because we believe the semantics of a syntactic unit should be independent of its syntactic context and only related to itself and its semantic context like its current thread. Based on this insight, HCSPs and UMLs should not be regarded as *common* because they do not contribute to state transitions and are not effectful semantic contexts.

- *Syntax Productions*. Ideally, *common* could consist of left-hand syntax productions minus the *missing*. Yet, this approach is computationally intensive. To simplify, each rule in *common* is assigned a unique ID. This facilitates the deterministic application of backward rewrite rules, ensuring the recovery of missing syntax productions when common variables differ. However, this approach introduces a limitation: it can result in indistinguishable syntax productions on the left side in scenarios where missing syntax productions exist, and common variables are identical across different rewrite rules. Despite this, it is reasonable because identical matching logic symbols should have the same meaning, given that the semantics of matching logic is formal and unambiguous. For example, in Fig. 5, if there is another rule with a right-hand pattern logically equivalent to lines 4-5, and the complements contain the same A and P , then the backward transformation generated according to Section 4.3 would result in non-deterministic application of the rewrite rule. This essentially means that multiple HCSP symbols correspond to a single PlantUML symbol, making it impossible to deduce which HCSP symbol should be mapped to when the PlantUML symbols are identical. This is reasonable because lines 1 and 6 in Fig. 1 should not have two different meanings. A simple modification to the arrow's annotation or color can differentiate them. This semantic consistency is ensured through lines 8-9 and 15 of Fig. 9 and lines 8-9 and 15 of Fig. 10.
- *Other information types.* State productions and rule attributes, similar to variables common in the same state, are excluded in *common*. Side conditions, analogous to syntax productions, are efficiently managed using rule ID.

Consequently, the state structure is $common \mapsto missing$, where $common$ is a combination of a rule ID and common variables/tokens across different states, and $missing$ is different variables and tokens. This structure's generality lies in its reliance on information from the language-agnostic $\mathbb{K}$ framework rather than specific transformation definitions. The rationale for this structure is further explored in Section 4.3, where we discuss backward rewrite and synthesis.

Workflow of Forward Transformation Synthesis. Figure 8 depicts the detailed synthesis workflow of KBX. We illustrate each step using Fig. 9, focusing on forward synthesis. The corresponding steps for backward synthesis are covered in Section 4.3, and step 8 is elaborated in Section 4.4. This example showcases a segment of the forward transformation definition Γ^{putr}, derived from the unidirectional Γ^{ux} in Fig. 5. The "**configuration**" keyword (line 1) sets up State+C , and the "**rule**" keyword (lines 4 to 7 and 9 to 12) defines rules in CreateR and PutR, respectively. A key difference in Fig. 9 is the *complements manipulation* for recovering missing information.

Step 1: $(syntax, state, rules) \leftarrow Extract(Parse(\Gamma^{ux}))$. The first step involves parsing the unidirectional transformation Γ^{ux} into KAST (K Abstract Syntax Tree), an intermediate representation in the $\mathbb{K}$ framework. This parsing, aided by $\mathbb{K}$, ensures the transformation targets' syntax, like HCSP's "_ ::= _", is intuitively represented. Then, we extract critical components from the KAST: $syntax$ (syntax definition, e.g., Fig. 5 lines 1-2), $state$ (transformation state declaration, e.g., Fig. 5 line 3), and $rules$ (rewrite/semantic rules, e.g., Fig. 5 lines 4-6).

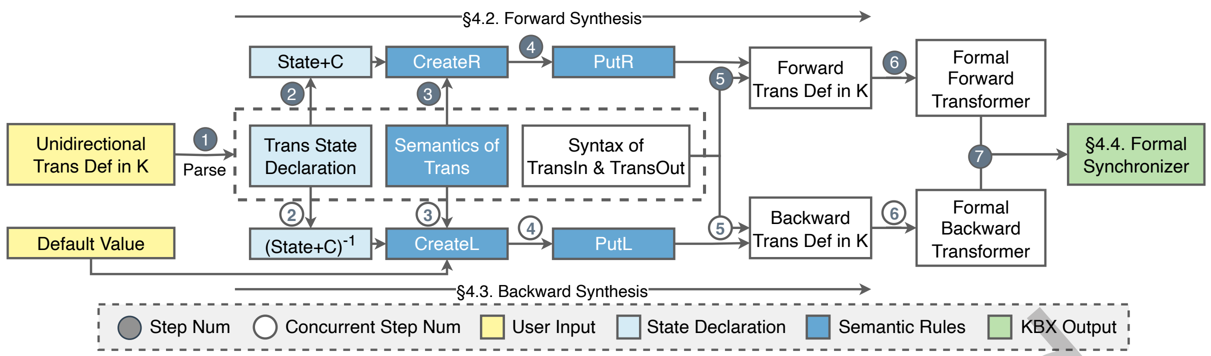


Fig. 8. The Detailed Synthesis Workflow of KBX Formal BX Framework.

```
1   // State+C
2   configuration <m> $PGM:HCSP </m> <n> .K </n> <s> .K </s> <c> .Map </c>
3   // CreateR: Complements Manipulation with Default Value
4   rule <m> [log( A ), L := R ] HCSPs : List => HCSPs </m>
5        <n> UMLs : List => UMLs [ P –[#black]> P : A ] </n>
6        <s> P </s>
7        <c> Cp:Map => Cp [[1, A , P ] <– [[ L , R ], [#black ]]]</c>
8    require Cp[[1, A , P ]] orDefault .List ==K .List
9            orBool Cp[[1, A , P ]] orDefault .List ==K [[ L , R ], [#black]]
10   [ priority (51) ]
11   // PutR: Complements Manipulation with Complements Holder
12  rule <m> [log( A ), L := R ] HCSPs : List => HCSPs </m>
13       <n> UMLs : List => UMLs [ P –[ C ]> P : A ] </n>
14       <s> P </s>
15       <c> Cp:Map [1, A , P ] |-> [[ _ , _ ], [ C ]]
16        => Cp:Map [1, A , P ] |-> [[ L , R ], [ C ]] </c>
```

Fig. 9. The $\mathbb{K}$ definition for the forward transformation of rule 1.

Step 2: $state^{+c} \leftarrow AddCHolder(state)$. This step adds a complements holder ($AddCHolder$) to the state for the forward transformation, differentiating it from the unidirectional state declaration. The complements holder, a *common* $\mapsto$ *missing* "Map", stores and retrieves missing information during the transformation. As shown in Fig. 9 line 2, $AddCHolder$ automatically generates the complements-holder cell with a distinct name "c" and its initial state ".Map".

Step 3: $create^{r} \leftarrow \forall\varphi\colon rule \in rules.\ Consist(\varphi\ CH(C \Rightarrow C(\varphi.common) \leftarrow (\varphi.miss_r, \varphi.miss_l)))$. This step involves enhancing each rewrite rule in the unidirectional transformation Γ^{ux} with complements manipulation. The process, applied to all rules ($rules$), generates new rules ($create^{r}$) that mirror the original rules but include the storage of complements. Specifically, the CH function is employed to create a complements-holder pattern. This pattern dictates the rewriting of the complements holder's content (C) to capture both $\varphi.miss_r$ (variables/tokens unique to the left-hand side) and $\varphi.miss_l$ (unique to the right-hand side). If no *missing* information (i.e. $\varphi.miss_r$,

$\varphi.miss_l$) is detected, the original rule is retained unchanged. Additionally, the *Consist* function ensures no complement conflicts in the transformation source by verifying the state of $C(\varphi.common)$. To achieve this, we introduce a precondition to the rewrite rules in $create^r$. This precondition checks whether the set $C(\varphi.common)$ is either empty or if $C(\varphi.common)$ is equivalent to the *missing* of the given rewrite rule, denoted as $C(\varphi.common) = () \vee C(\varphi.common) = (\varphi.miss_r, \varphi.miss_l)$.

Fig. 9 illustrates our approach in lines 4-10, introducing complements manipulation to Fig. 5 in lines 4-6. In line 7, a complements-holder state cell "<c>" is utilized, driven by the *CH* function. This cell involves a rewrite pattern whose left-hand side is any possible complements holder C (i.e., "Cp:Map"). The right-hand side is a map update operation ("<-"). The map updates based on the key: $\varphi.common$, i.e., a list of rule ID ("1") and common variables/tokens (A , P). The updated value is a list of $\varphi.miss_r$ (i.e., L and R) and $\varphi.miss_r$ (i.e., "#black"). The *Consist* function generates the side condition in lines 8-9. Line 10 introduces a lower priority for rule application, which is an implementation compromise for *HP* in the next step.

Note that, C (e.g., "Cp") can be any holder without impacting the application of rule φ. Thus, the generated rewrite rules $create^r$ defines an equivalent transformation to the original *rules*.

Step 4: $put^r \leftarrow \forall\varphi \in rules.\ let\ \varphi' \leftarrow T2V(\varphi)\ in\ HP(\varphi'\ CH(C(\varphi.common) = (Any, \varphi'.miss_l) \Rightarrow C(\varphi.common) = (\varphi'.miss_r, \varphi'.miss_l)))$. This step generates rewrite rules PutR (put^r), using the complements holder for information recovery. This recovery is achieved by the holder's left-hand side $\varphi'.miss_l$. This rule also supports the modification of the synchronization source through *Any* and only uses $\varphi.common$ as the identification for obtaining missing information. This synthesis step involves two new functions: *T2V* and *HP*. The *T2V* function converts all right-hand tokens into variables to use the complements holder for information recovery, rather than the default values. The *HP* function assigns these new rules a higher priority, ensuring they are applied before $create^r$ when matched complements exist.

Fig. 9 demonstrates the generation of put^r in lines 12-16. This example mirrors Fig. 5 (lines 4-6), except for the color token "#black" is changed to the variable C using *T2V*. The complements-holder cell "<c>" is then generated according to the new pattern φ' using *CH*. As discussed before, the *HP* function provides a lower priority to the rules in $create^r$ to ensure the precedence of put^r. This compromise is because the default priority in $\mathbb{K}$ is 50, which is also the highest priority.

Note that, the rules in put^r are not applicable when complements are empty. In such cases, the behavior of the transformation *putr* follows that of *ux* as per the rules in $create^r$.

Step 5: $\Gamma^{putr} \leftarrow Print(syntax, state^{+c}, create^r, put^r)$. This step involves pretty printing the modified KAST into a readable $\mathbb{K}$ definition for Γ^{putr}.

Step 6: $putr \leftarrow Kompile(\Gamma^{putr})$. The final step is compiling the Γ^{putr} definition into a verifiable and executable forward transformer using *Kompile*.

In summary, the forward synthesis fulfills the following forward transformation requirements: (1) When complements are absent, the forward transformation *putr* mirrors the unidirectional transformation *ux* using rules in CreateR ($create^r$). (2) With complements, *putr* uses PutR ($putr^r$) to recover missing information. (3) All missing information *missing* is retained in the complements holder $common \mapsto missing$, identifiable through *common* between rewrite rules. (4) Transformation *putr* validates the consistency of missing information in the transformation source using preconditions generated by *Consist*. In essence, irrespective of whether the complements holder has retained the information, *putr* maintains the same *missing*.

4.3 Backward Transformation Synthesis

Based on the complements holder and manipulation strategy, this section presents the synthesis basis and workflow for the backward transformation *putl*.

Theory of Backward Rewrite: To accurately synthesize backward transformation definition Γ^{putl}, we introduce the concept of *backward rewrite* (denoted as $(\_)^{-1} \in \Sigma$) within matching logic to achieve a stringent recovery formalized by the BackForth law.

Definition 7. Given $\varphi \equiv \varphi_{lhs} \Rightarrow \varphi_{rhs}$, we define its backward rewrite rule, denoted as $(\varphi)^{-1}$ (abbreviated for the application $(\_)^{-1}\varphi$), as follows:

$$(\varphi)^{-1} \equiv \varphi_{rhs} \Rightarrow \varphi_{lhs}$$

This definition shows that the backward rewrite $(\_)^{-1}$ logically exchange φ_{lhs} and φ_{rhs} within the rewrite rule φ. Aggregating these backward rewrite rules forms the backward semantics Γ^{-1}, enabling the recovery of initial states from final states for forward semantics Γ. This is specified by the BackForth law, given below:

Definition 8. Given any rewrite rule $\varphi_{lhs} \Rightarrow \varphi_{rhs} \in \Gamma$, we define the backward semantics as $(\Gamma)^{-1}$ as follows:

$$(\varphi_{lhs} \Rightarrow \varphi_{rhs})^{-1} \in (\Gamma)^{-1}$$

This definition adheres to the back-forth law, which holds for all states φ_{init} and φ_{final}:

$$\frac{\Gamma \vdash \varphi_{init} \Rightarrow \varphi_{final}}{(\Gamma)^{-1} \vdash \varphi_{final} \Rightarrow \varphi_{init}} \qquad \text{(BackForth)}$$

The satisfaction of the BackForth law can be understood through a two-step process. Firstly, we break down the forward execution, denoted as $\Gamma \vdash \varphi_{init} \Rightarrow \varphi_{final}$, into discrete sequential steps from φ_i to φ_{i+1}. At each step φ_i, we identify a corresponding rewrite rule $\varphi_{lhs} \Rightarrow \varphi_{rhs}$ within Γ and utilize a substitution θ_i such that $\varphi_{lhs}\theta_i \equiv \varphi_i$, thereby progressing to the subsequent step φ_{i+1}. Secondly, we initiate the backward execution commencing with the final state φ_{final}. At each subsequent step φ_{i+1}, we identify a corresponding backward rewrite rule $(\varphi_{lhs} \Rightarrow \varphi_{rhs})^{-1}$ within the backward semantics $(\Gamma)^{-1}$ and employ a substitution θ_{i+1} such that $\varphi_{rhs}\theta_{i+1} \equiv \varphi_{i+1}$, effectively regressing to the previous step φ_i. The construction of the backward execution, denoted as $(\Gamma)^{-1} \vdash \varphi_{final} \Rightarrow \varphi_{init}$, takes place step by step. Specifically, it is not necessary to present the reversed version of BackForth like the round-tripping laws in Def. 1 since $((\Gamma)^{-1})^{-1} = \Gamma$.

Workflow of Backward Transformation Synthesis. Leveraging the logical recovery potential of the backward rewrite, we follow Figure 8's steps to synthesize $putl$. Steps 1-2 are similar to forward transformation synthesis (discussed in Section 4.2) and focus on subsequent steps. To clarify, we provide the following segment from the $\mathbb{K}$ definition for the backward transformation.

Step 2: $(state^{+c})^{-1} \leftarrow ReverseIO(state^{+c})$. In this step, we utilize the state declaration $state^{+c}$ from the forward synthesis and apply *ReverseIO* to interchange the input and output cells. This interchange modifies their initial states without altering the overall configuration structure for rewrite rules. For example, as demonstrated in Fig. 10 (line 2), *ReverseIO* identifies the input cell ("<m>") by its content, labeled as "$PGM", which is then denoted as the terminal symbol ".K". Concurrently, the output cell ("<n>") is assigned the content "$PGM" of type "K", which denotes any type defined in $\mathbb{K}$ including the syntax of the transformation target.

In line with the back-forth law (BackForth), the unidirectional transformation ux should conclude with $(state^{+c})^{-1}$ to ensure accurate recovery. Therefore, developers must ensure that the transformation ux concludes with its initial state $state$, except the input cell becomes the terminal symbol ".K", and the output cell is adaptable to any model with type "K". Step 6 offers a modifiable definition, affording developers the flexibility to make alterations to $(state^{+c})^{-1}$ and thereby eliminating constraints on the final state of ux.

Step 3: $create^{l} \leftarrow \forall \varphi\colon rule \in rules.\ F^4(VT2M(\varphi^{-1}))$. This step enhances each rewrite rule in Γ^{ux}, integrating backward rewriting and complements manipulation. Firstly, we apply the backward rewrite, $(\_)^{-1}$, to each rule φ, creating its backward. In Fig. 10 (lines 4-6, 12-15), this is shown as a reversed rewrite pattern within cells <m>, <n>, and <s>. This reversal also includes the contents of **require** and **ensure**, as well as ensuring consistency

```
1  // (State +C)^-1
2  configuration <m> .K </m> <n> $PGM:K </n> <s> .K </s> <c> .Map </c>
3  // CreateL: Complements Manipulation with Default Value
4  rule <m> HCSPs => [log( A ), ?1? := ?2?] HCSPs </m>
5       <n> UMLs [ P -[ C ]> P : A ] => UMLs </n>
6       <s> P </s>
7       <c> Cp:Map => Cp [[1, A , P ] <- [[ ?1? , ?2? ], [ C ]]]</ c>
8  require Cp[[1, A , P ]] orDefault .List ==K .List
9          orBool Cp[[1, A , P ]] orDefault .List ==K [[ ?1? , ?2? ], [ C ]]
10 [ priority (51) ]
11 // PutL: Complements Manipulation with Complements Holder
12 rule <m> HCSPs => [log( A ), L := R ] HCSPs </m>
13      <n> UMLs [ P -[ C ]> P : A ] => UMLs </n>
14      <s> P </s>
15      <c> Cp:Map [1, A , P ] |-> [[ L , R ], [ _ ]]
16       => Cp:Map [1, A , P ] |-> [[ L , R ], [ C ]] </c>
```

Fig. 10. The $\mathbb{K}$ definition for the backward transformation of rule 1.

and standardization of variables within the rule. Secondly, the *VT2M* function transforms variables and tokens, appearing only on the right side of "=>", into question marks with unique identifiers. As demonstrated in Fig. 10 line 4, variables L and R become ?1? and ?2?, respectively. This conversion enables users to define their Default Value for unknown missing information. Lastly, F^4 function mirrors the operation in Step 4 of forward synthesis, as shown in the following desugared format:

$$create^l \leftarrow \forall \varphi : rule \in rules.\text{let } \varphi \leftarrow VT2M(\varphi^{-1}) \text{ in}$$
$$Consist(\varphi\ CH(C => C(\varphi.common) \leftarrow (\varphi.miss_r, \varphi.miss_l)))$$

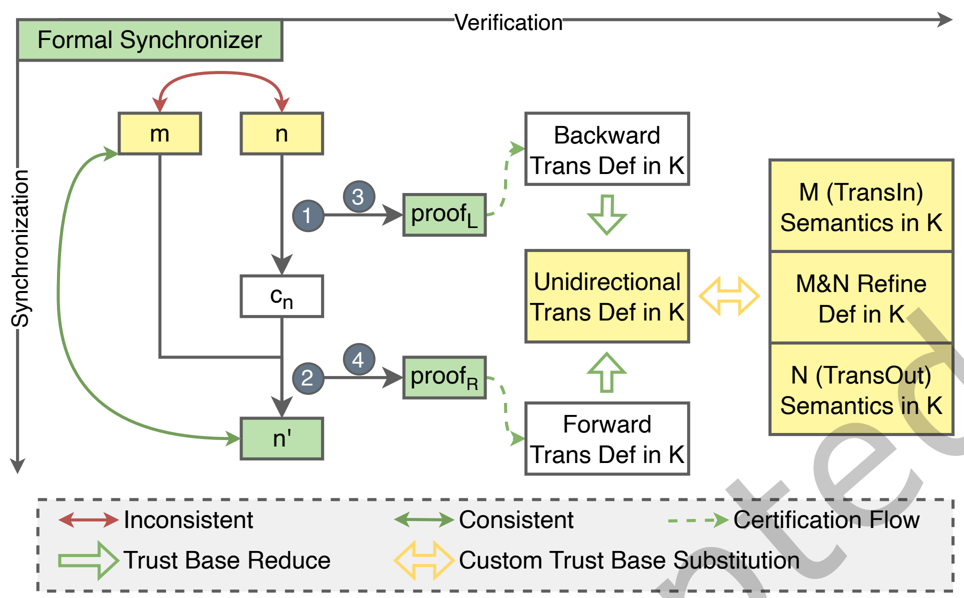
Note that, the interpretation of $\varphi.miss_r$ and $\varphi.miss_l$ is relative to the unidirectional transformation ux direction. In the backward synthesis context, $\varphi.miss_r$ represents variables/tokens unique to the right-hand side, and $\varphi.miss_l$ to those unique to the left-hand side. Fig. 10 lines 4-10 showcases the rule generated in this step.

Step 4: $put^l \leftarrow ExAny((put^r)^{-1})$. In this step, we apply the backward rewrite to each rule in put^r, creating their backward counterparts in put^l. The *ExAny* function exchange the postion of Any on the left-hand side of put^r, obtaining $(\varphi'.miss_r, Any)$ for put^l. This process ensures complete adherence to the BackForth, enabling a seamless transition between states in the transformation trace for recovery. The application of this step is exemplified in Fig. 10 (lines 12-15).

Step 5: $\Gamma^{putl} \leftarrow Print(syntax, (state^{+c})^{-1}, create^l, put^l)$. This step prints the modified KAST into a $\mathbb{K}$ definition for Γ^{putl}.

Step 6: $putl \leftarrow Kompile(\Gamma^{putl})$. Finally, Γ^{putl} is compiled into an executable backward transformer $putl$ using $Kompile$.

In summary, these steps yield $putl$, the backward transformation, which, in conjunction with $putr$, achieves formal bidirectional transformation for simultaneous consistency verification and synchronization. The round-tripping laws' satisfaction will be discussed in the following section.



Let m and n respectively represent an HCSP program and a UML model, with c_n denoting missing UML information in HCSP (e.g., color). This information is extracted as per the $\mathbb{K}$ definition Γ^{putl}, verified by $proof_L$. A synchronized UML model, n', is achieved, whose consistency with m is confirmed by $proof_R$. Consistency between HCSP (M) and UML (N) is defined by Γ^{ux}, inherited by definitions Γ^{putr} and Γ^{putl}.

Fig. 11. The Workflow of Formal Synchronizer.

4.4 Formal Synchronizer for Simultaneous Synchronization and Verification

This section introduces a formal synchronizer in the KBX framework, leveraging transformers $putr$ and $putl$. This synchronizer ensures simultaneous synchronization and verification of consistency relation R between models m and n, backed by formal proofs. It also aligns with the round-tripping laws (Definition 1) to guarantee reasonable synchronization and establishes an equivalence between R and refinement relation R_{ref} through forward simulation. The workflow, including both synchronization and consistency verification processes, is depicted in Fig. 11.

Reasonable Synchronization Satisfying Round-tripping Laws. The synchronizer inputs two models m and n and outputs synchronized models m' and n'. Fig. 11 illustrates a typical synchronization scenario where m remains unchanged ($m = m'$), and n is adjusted to align with m. The synchronizer is versatile, allowing synchronization in both directions due to the commutative nature of $putr$ and $putl$. Step 1 involves applying $putl$ to model n, extracting missing information c_n relative to model m. This step primarily uses rules from $create^l$ as no information in the complements holder matches rules in put^l. Step 2 employs $putr$, incorporating c_n into the complements holder, to produce the consistent model n'. This step relies on rules in put^r, where missing information is available in the complements holder.

These two steps are integrated into $putr$ as outlined in Definition 1, and reversing their order results in $putl$, ensuring compliance with the round-tripping laws. To elaborate, upon completing these steps, the output is (n, c') as per PutLR. Applying the transformer $putl$ to this output yields the same (m, c'). The complements holder remains unchanged since steps 1 and 2 have already captured the information from both m and n, and the rules in both $create^r$ and put^r preserve identical information for the same source. Consequently, the original m

is retrieved using only the rules in put^l. The rules in Γ^{putr} and Γ^{putl} adhere to the BackForth law, barring the complements holder. For instance, regardless of whether a rule is in $create^r$ or put^r, applying put^l retrieves the same left-hand side, leading back to m from (n, c'). Since the BackForth law is directionally neutral, this process is reversible, ensuring the round-tripping laws are fully met.

Reliable Consistency Verification Comparable to Refinement Verification. Steps 3-4 in the consistency verification process verify the consistency between m and n'. Step 3 generates proof $proof_L$, confirming that the generation of c_n aligns with Γ^{putl} rules. Step 4 produces proof $proof_R$, verifying the consistency of n' with m as per the $\mathbb{K}$ definition Γ^{putr}. Both proof generation processes follow the methodologies presented in [Chen et al. 2021a]. These processes involve initially producing proof hints during transformation. Subsequently, we leverage these hints to construct Metamath proofs. These proofs encompass axioms derived from the $\mathbb{K}$ definition, a proof goal that validates the correctness of the rewriting from φ_{init} to φ_{final}, and a demonstration using the matching logic proof system and axioms to substantiate this goal. We then employ a Metamath checker to confirm that the generated proof satisfactorily resolves the proof goal. Proof generation, independent of transformer application order, is grounded in Γ^{ux} and synthesis algorithms, forming the formal definition of consistency relation R.

Definition 9. The consistency relation R between arbitrary models $m : M$ and $n : N$ is defined as follows,

$$(m, n) \in R \equiv \Gamma^{putr} \vdash m\ \sigma_n\ \sigma_c\ \sigma_s \Rightarrow \sigma_m\ n'\ c\ \sigma_s \wedge \Gamma^{putl} \vdash n\ \sigma_m\ c\ \sigma_s \Rightarrow \sigma_n\ m\ c'\ \sigma_s$$

or equivalently,

$$(m, n) \in R \equiv \Gamma^{putl} \vdash n\ \sigma_m\ \sigma_c\ \sigma_s \Rightarrow \sigma_n\ m'\ c\ \sigma_s \wedge \Gamma^{putr} \vdash m\ \sigma_n\ c\ \sigma_s \Rightarrow \sigma_m\ n\ c'\ \sigma_s$$

The relation R between models m and n has two equivalent definitions based on the application order of Γ^{putr} and Γ^{putl}. In both of these definitions, the left side of the conjunction is employed for acquiring complements based on Γ^{ux}, while the right side of the conjunction ensures the consistency defined within Γ^{ux}. Intuitively, the first definition represents a matching logic proof where $putr(m, \emptyset) = n', c \wedge putl(n, c) = m, c'$ from the perspective of a symmetric lens. The second definition is the reverse: $putl(n, \emptyset) = m', c \wedge putr(m, c) = n, c'$.

Our consistency verification, based on these definitions, ensures both syntactic consistency (models conform to a specific syntax) and semantic consistency (models have the same meaning), similar to refinement verification. The consistency definition Γ^{ux} should have correct syntax and semantics for synchronization targets, just as formal semantics must be accurate for refinement verification. If the syntax is incorrect, the verification will fail because the incorrect syntax is not defined in matching logic theory Γ^{ux}. If the semantics (both static and dynamic) are incorrect, there is no rule to match the incorrect state and transition the input model to the output model. For example, if there is only a send message in the HCSP program, the transformation process will be stuck in a deadlock, preventing the achievement of an empty state σ_s to conclude the transformation. To illustrate this semantic consistency, the refinement relation R_{ref} is defined similarly, but with operational semantics Γ^M and Γ^N.

Definition 10. The refinement relation R_{ref} between arbitrary models $m : M$ and $n : N$ is defined as follows:

$$\begin{aligned}(m, n) \in R_{refine} \equiv\ & (\Gamma^M \vdash m\ \sigma_{s_m} \Rightarrow \sigma_m\ s_m) \\ \wedge\ & (\Gamma^N \vdash n\ \sigma_{s_n} \Rightarrow \sigma_n\ s_n) \\ \wedge\ & R_S\ \sigma_{s_m}\ \sigma_{s_n} \wedge R_S\ s_m\ s_n\end{aligned}$$

Here, Γ^M and Γ^N are the operational semantics for models m and n, while s_m and s_n represent the execution states for these semantics. σ_{s_m} and σ_{s_n} refer to the initial states of m and n, respectively, and R_S defines the state relation predicate pattern. The semantics Γ^M and Γ^N are defined as follows: $\forall m{:}M, n{:}N.\ \exists s_m{:}S_M, s_n{:}S_N.$

$$\Gamma^M \vdash m\ \sigma_{s_m} \Rightarrow \sigma_m\ s_m \qquad\qquad \Gamma^N \vdash n\ \sigma_{s_n} \Rightarrow \sigma_n\ s_n$$

Hence, there are three types of compliance relations between consistency relation R and refinement relation R_{ref}: (1) R is an equivalent of R_{ref} (written $R = R_{ref}$); (2) R is a refinement of R_{ref} (written $R \subset R_{ref}$); (3) R_{ref} is a refinement of R (written $R_{ref} \subset R$). Table 1 compares KBX-based consistency verification and refinement verification for models m and n.

Table 1. Comparison of KBX-based Verification and Refinement Verification.

	KBX	Refinement
Trust Base	Γ^{putr}, Γ^{putl} (or Γ^{ux}, Synthesis)	Γ^{M}, Γ^{N}, R_S
Applicable	Complex M,N; Simple M,N Relation	Simple M,N; Complex M,N Relation
Method	Verified Synchronization	Symbolic Execution or Theorem Proof
On Failure	Automatic Alignment	Manual Alignment

In summary, KBX provides a formal synchronizer that facilitates reasonable synchronization and reliable consistency verification between models m and n.

5 EVALUATION

We have implemented the proposed approach in the KBX tool, which is built on top of the $\mathbb{K}$ formal framework. To assess the effectiveness of our KBX, we address the following research questions:

- **RQ1**: How does the development efficiency and trustworthiness of KBX compare to the existing BX frameworks?
- **RQ2**: How does KBX perform in terms of both its expressiveness and trustworthiness as a verification approach?
- **RQ3**: How can KBX facilitate the development of formal BX between UML and concurrent HCSP to address our industrial scenarios?

5.1 RQ1: Comparison with Existing BX Framework

In this section, we compare KBX against other BX frameworks using the criteria of development efficiency(E) and trustworthiness(T) in Table 2. Concerning [Anjorin et al. 2020; Buchmann et al. 2022], we choose typical and relevant BX frameworks for comparison, including BiGUL [Bettini and Efftinge 2016; Ko et al. 2016], BiYacc [Zhu et al. 2015], JTL [Cicchetti et al. 2011], eMoflon [Weidmann et al. 2019], NMF [Hinkel and Burger 2019], EVL+Strace [Samimi-Dehkordi et al. 2018], BXtend [Buchmann 2018]+BXtendDSL [Buchmann et al. 2022], and Hobit [Matsuda and Wang 2018]+Synbit [Yamaguchi et al. 2021]. Specifically, we present the following assessment questions to qualify for the industrial requirements:

E: The capability of constructing BX efficiently:
- **E1:** Can we construct symmetric BX without implementing two asymmetric BX programs?
  - *Rationale:* Missing information on both sides of synchronization necessitates two asymmetric lenses, raising implementation costs and increasing the likelihood of errors in synchronizer development and maintenance.
- **E2:** Can we construct BX in consistency definitions (e.g., unidirectional transformation definitions) without implementing the BX programs?
  - *Rationale:* Although every programming language can implement BX, the consistency definition streamlines this by merging two-sided transformation and complement concepts into one representation.

Table 2. The satisfaction of BX frameworks in terms of **E**xpressiveness (E) and **T**rustworthiness (T).

	KBX	BiGUL	BiYacc	JTL	eMoflon	NMF	EVL+	BXtend	Hobit
E1	🟩 §4.1	🟥 3	🟥 6	🟩 7	🟩 3	🟩 9	🟩 3	🟩 11	🟥 12
E2	🟩 §4	🟩 4	🟩 6	🟩 7	🟩 3	🟩 9	🟩 10	🟩 11	🟩 13
E3	🟩 1	🟥 5	🟩 6	🟥 3	🟥 3	🟥 3	🟥 3	🟥 11	🟥 12
T4	🟩 §4	🟥 5	🟥 6	🟥 8	🟥 3	🟥 3	🟥 3	🟥 11	🟥 12
T5	🟩 1	🟥 5	🟥 6	🟥 8	🟥 3	🟥 3	🟥 3	🟥 11	🟥 12
T6	🟩 §4.1	🟩 4	🟩 6	🟩 8	🟩 3	🟩 9	🟥 3	🟩 11	🟩 12
T7	🟩 2	🟥 5	🟥 6	🟥 8	🟥 3	🟥 3	🟥 3	🟥 3	🟥 12

The green color in the table indicates satisfaction, while the red color signifies non-satisfaction. The numbers or section numbers in the table provide the corresponding evidence for these conclusions.

1 [Chen et al. 2021a]; 2 $\mathbb{K}$ Framework; 3 [Anjorin et al. 2020]; 4 [Ko et al. 2016]; 5 [Hu and Ko 2018]; 6 [Zhu et al. 2020]; 7 [Cicchetti et al. 2011]; 8 [Eramo et al. 2018]; 9 [Hinkel and Burger 2019]; 10 [Samimi-Dehkordi et al. 2018]; 11 [Buchmann et al. 2022]; 12 [Matsuda and Wang 2018]; 13 [Yamaguchi et al. 2021]

E3: Can we construct BX with BNF-like grammar without implementing parsers, printers, etc?
- *Rationale:* Given that our intended synchronization targets are models or programs in various languages, the BX framework should automatically generate synchronizers that can manage these targets efficiently without additional effort.

T: The capability of trustworthiness for verification:

T4: Can we define formal correctness specification for bidirectional transformation directly?
- *Rationale:* If the answer is affirmative, BX developers can formally verify their BX programs, eliminating the implementation and trust in translating into formal frameworks.

T5: Can we verify the formal specification during the synchronization?
- *Rationale:* This guarantees rigorous adherence of the synchronized targets to the user-defined formal specifications.

T6: Can we guarantee the BX behavior by explicit laws?
- *Rationale:* BX without an explicit rule lacks a clear definition of rational synchronization.

T7: Can we validate the execution of BX with a proof checker?
- *Rationale:* This question ensures that the synchronized targets adhere to the formal definition of consistency, which is reliable and can be readily verified by a third party.

Table 2 displays the BX frameworks' evaluation results, with each cell indicating a framework's response to an assessment question. A "Yes" response is green and a "No" response is red. Every response in the table refers to specific evidence, indicated by a corresponding number at the table's bottom. Under the same evaluation methodology, we present the relation between our evidence with the assessment question as follows.

E1: Section 4.1 captures the symmetric lenses. As noted in [Anjorin et al. 2020] and their own literature, JTL, eMoflon, NMF, EVL+Strace, and BXtend support handling missing information on both sides.
In contrast, BiGUL uses a putback-based method for asymmetric lenses. According to [Anjorin et al. 2020], it requires an intermediate representation and two BX programs to handle missing information on both sides. Similarly, BiYacc and Hobit rely on the asymmetric lens, thus they cannot handle missing information on both sides.

E2: Sections 4.2 and 4.3 introduce the bidirectional transformation synthesis algorithm. Other BX frameworks also provide bidirectional programming methods to generate BX programs.

E3: Section 4.1 takes 𝕂 specifications as input and output, while the 𝕂 framework can generate corresponding tools automatically. Additionally, [Chen et al. 2021a] presents a simple 𝕂 definition with BNF-like grammar. BiYacc is specifically designed to facilitate the simultaneous development of parsers and printers, supporting BNF-like grammar. However, its specialization likely limits its expressiveness. As [Kinoshita and Nakano 2017] demonstrates, it is necessary to combine BiYacc and BiGUL to achieve bidirectional transformation between two languages.
In contrast, other bidirectional transformation (BX) methods do not consider the generation of parsers and printers essential for a BX framework. This lack of integration results in less intuitive and concise definition of consistency between languages, such as directly mapping the grammars of the synchronization targets "log(A); L := R ;" to " P -[#black]-> P : A ". Furthermore, it necessitates additional code for implementing the synchronization targets' parsers and printers. This added complexity hampers the review of the trust base (i.e., consistency definition), making it more challenging to ensure trustworthiness. Consequently, the application of BX across different programming languages is limited.

T4: Section 4.1 shows that KBX's input is a 𝕂 definition that allows specifications for correctness like [Alpuente et al. 2020; Hathhorn et al. 2015]. Since the algorithms in Sections 4.2 and 4.3 focus on the complements recovery and don't modify the specification, the correctness specification also affects the bidirectional transformation.
Other methods do not describe BX using a formal language with an embedded proof system. BiGUL is the closest, having formalized its lens semantics in AGDA. However, its bidirectional programming language does not inherently possess a sound proof system, making it impossible to describe and prove correctness specifications for its synchronization targets.

T5: Section 4.1 takes 𝕂 specifications as input and output, while 𝕂 allows symbolic execution for verification using these specifications.
Unlike our approach, other BX frameworks lack formal verification methods, such as symbolic execution, model checking, or abstract interpretation, during synchronization. Moreover, we did not find any textual or code-based evidence of integrating SMT solvers, which would result in a large and unreliable trust base.

T6: Section 4.1 formalizes the round-tripping laws within matching logic. Sections 4.2 and 4.3, present algorithms to synthesize bidirectional transformations that align with these laws.
For general BX frameworks, correctness—or the guarantees defined in [Anjorin et al. 2020]—is the satisfaction of the explicit laws. Consequently, this criterion is generally satisfied by all such frameworks. However, we did not find the definition of reasonable synchronization for EVL+Strace in either [Samimi-Dehkordi et al. 2018] or the survey [Anjorin et al. 2020].

T7: Sections 4.2 and 4.3 demonstrate KBX's ability to generate bidirectional transformation 𝕂 specifications. Subsequently, we can employ these specifications to verify the execution of BX automatically using the method proposed in [Chen et al. 2021a].
Some other BX frameworks provide methods to ensure synchronization meets user-defined specifications. However, without formal semantics (except for BiGUL), these user-defined specifications are informal and ambiguous. Even with formal semantics, no framework offers a formal verification method, as explained in T5. In summary, none rely on third-party proof checkers for high-assurance formal verificatio, nor even less reliable SMT solvers.

To intuitively illustrate KBX's efficacy in constructing BX, we employ the *Family and Person* BX benchmark introduced by [Anjorin et al. 2020]. This benchmark involves maintaining consistency between two models with missing information on both side: (1) a families model with a list of families, each containing a family name and members with their first names and roles, and (2) a persons model with a flat list of names and edges. Table 3 presents the definition size statistics for different BX frameworks.

Table 3. Size of the transformation definitions of all solutions.

	KBX	BiGUL	BiYacc	JTL	eMoflon	NMF	EVL+	BXtend	Hobit
LOC	32	176	-	227	217	279	1299	211	-
N words	312	1010	-	773	337	607	2878	565	-

KBX synthesizes 2491 words definition for bidirectional transformation.

The table presents several metrics, including Lines of Code (LOC) and word counts (N words), with a ”-” indicating missing data. The data reveals that constructing BX with KBX requires *the least human effort*. This benchmark exclusively employs JSON as both input and output formats and omits the line of code count for the parser and printer, which constitutes a significant portion. Indeed, the generated definition extends to 2,459 words, highlighting the substantial labor involved in implementing BX within the $\mathbb{K}$ framework without the assistance of KBX.

In summary, despite graphical limitations such as the absence of a graph language and interface, **KBX surpasses other BX frameworks for a specific set of queries**.

5.2 RQ2: Capability as a Verification Approach

In this section, we shift our attention from KBX synchronization capabilities to its formal verification potential. We will compare KBX with existing verification methods to answer two crucial questions:

- **RQ 2.1**: What level of verification can we achieve through KBX expressiveness?
- **RQ 2.2**: How trustworthy are the outcomes of this verification process?

RQ 2.1: We first evaluate the consistency definition (i.e., input) of KBX by revealing its expressiveness as a correctness specification step by step.

First, we consider the theoretical maximum expressiveness based on matching logic, which captures many important logics in mathematics and computer science, including first-order logic with least fixpoints, modal μ-logic, separation logic, dynamic logic, various temporal logics and reachability logic [Chen and Roşu 2019]. Notably, verification using reachability logic is no harder than using Hoare logic [Stefănescu et al. 2016]. Hoare logic is widely used in theorem-proving-based program verification, such as the renowned seL4 [Klein et al. 2014, 2010].

Second, based on matching logic, the $\mathbb{K}$ framework provides language-agnostic verification through reachability logic for languages with formal semantics defined in $\mathbb{K}$ [Stefănescu et al. 2016]. This includes languages such as C [Ellison and Rosu 2012], Java [Bogdanas and Roşu 2015], JavaScript [Park et al. 2015], and x86-64 [Dasgupta et al. 2019]. Consequently, KBX, which is built upon the $\mathbb{K}$ framework, inherits these verification capabilities.

Lastly, we evaluate whether our input restrictions affect the expressiveness for correctness specification. The expressiveness is considered unrestricted when individual rewrite rules, the fundamental unit for specifying correctness, are unconstrained. Furthermore, we argue that overall semantic restrictions do not compromise expressiveness but instead refine a specific set of semantics, akin to classifying a language feature as object-oriented. We review all definitions and algorithms presented in the paper:

In Section 4.1: (1) Definition 4 characterizes the input as a model transformation $M \rightarrow N$ that supports dynamic semantic descriptions, enabling state model S. In contrast, general formal semantics are represented as Γ^M or Γ^N in Section 4.4. Thus, Definition 4 refines S_M into the transformation target N and the state model S needed for the transformation. This is similar to considering state cells <n> and <s> (in Fig. 5) as both S_M or more precisely as NS. Furthermore, the additional requirement of Definition 4 is that <n> must be a PlantUML symbol after the transformation. This not only preserves expressiveness but also precisely defines the transformation target as a program conforming to PlantUML syntax. (2) Definitions 5 and 6 specify the KBX-generated definitions, without affecting user-provided consistency definition.

The algorithms in Sections 4.2 and 4.3 manipulate each rewrite rule of the consistency definitions but do not impose restrictions on how users write them. Instead, we construct correctness specifications and generate preconditions, such as those on lines 8-9 in Fig. 9, to avert inconsistencies between the transformation source and stored complements. Additionally, the limitations mentioned in Section 4.2 are not limitations on user-defined syntax but design choices for the complements holder: default order sensitivity or not. If we record order information, lines 3-4 and 8-9 in Fig. 6 can be interpreted as distinct semantic elements, corresponding to different arrow colors. However, in our current design, they can only correspond to one arrow color based on generated preconditions. Beyond the reasons mentioned in Section 4.2, we contend this is reasonable as they possess the same semantics and thus must maintain the same semantics in UML. This does not diminish expressiveness for verification.

This design even affords greater flexibility for BX, as this order information can be constructed within the consistency definition, akin to placing the HCSP process identifier in <s> (line 6 in Fig. 5). Implementing this semantics is simpler because it is not a concurrent semantic context like the HCSP process. Conversely, designing the complement holder as order-sensitive (e.g., a list of $(common, missing)$ replacing $common \rightarrow missing$) would impede the construction of order-independent BX and prevent automatic verification of non-ambiguity semantics in synchronized targets. Specifically, it would fail to ensure that the semantics of lines 3-4 and 8-9 in Fig. 6 are unambiguous in UML, i.e., ensuring that all UML line colors are green.

To summarize RQ 2.1, the expressiveness of KBX for verification can be intuitively described as shown in Fig. 12: KBX utilizes the $\mathbb{K}$ framework without limitations on its expressiveness for verification. The $\mathbb{K}$ framework employs reachability logic, which is comparable to Hoare logic, widely used in theorem-proving-based program verification.

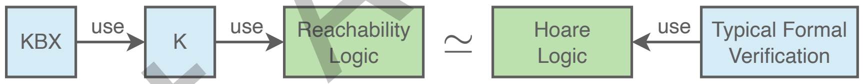


Fig. 12. Expressiveness Comparison for Verification

RQ 2.2: To assess trustworthiness, we examine the trust base of KBX—a critical factor necessary for establishing confidence in the verification outcome.

Table 4. Comparison of Trust Base

Approach	Consistency Definition	Trusted Kernel	Other Trust Base
KBX	Γ^{ux}	245 + 74	1015
$\mathbb{K}$	Γ^M, Γ^N, R_S	Same as above	-
Isabelle	Same as above	2152	Translation to Isabelle
Coq	Same as above	2666	Translation to Coq
Lean	Same as above	6132	Translation to Lean

Table 4 presents the trust bases for different formal frameworks, each aimed at verifying the semantic consistency between two models, $m \in M$ and $n \in N$. For each method, the trust base comprises the consistency definition, trusted kernel for verification, and other trust base. For KBX, the consistency definition employs a unidirectional transformation definition from M to N denoted as Γ^{ux}. The trusted kernel consists of 245 lines of Matching Logic formalization [Chen et al. 2021a] and 74 lines of Mathematica code implementing the proof checker[13]. Additionally, the other trust base includes 1015 lines of KBX implementation code or the formal definitions generated by this code. The $\mathbb{K}$ Framework employs formal semantics Γ^M, Γ^N and the state relations R_S between them, sharing the same trusted kernel as KBX, with no additional trust base. Isabelle, Coq, and Lean also utilize refinement verification, with their trusted kernels consisting of 2152 lines of ML code[14], 2666 lines of OCaml code[15], and 6132 lines of C++ code[16] respectively. When models are not within the formal framework, these methods require translation of M and N into their respective frameworks, which constitutes their additional trust base. In comparison to these tools with small trust base, typical SMT solvers like Z3 have a codebase exceeding 500,000 lines, indicating a significant difference in scale. Therefore, we assert that even trusting the KBX implementation still falls within the magnitude of theorem proving trust bases.

In summary, **KBX has the capability to verify model correctness, consistency, and refinement with a small trust base and minimal constraints on the expressiveness of $\mathbb{K}$.**

5.3 RQ3: Formal HCSP and UML BX for Industrial Scenarios

This section presents the first HCSP and UML BX, showcasing KBX's real-world effectiveness in addressing industrial challenges (Section 3.1). This BX enables automatic synchronization of the HCSP and UML models and verification of their consistency. During our collaboration with an industry partner, we focus on modeling and validating a critical function in maglev train systems: partition handover. The function, encompassing 13 system components and 29 message types, requires seamless and secure transitions of maglev trains between rail segments and communication partitions. This necessity arises from the constraints of limited rail length and wireless communication range. Due to the use of different modeling languages, the development and verification of multiple models is tedious and error-prone. For example, communication and synchronization take up most of the time during the five iterations to construct the 15 UML diagrams and 10 HCSP scenarios. Throughout these iterations, we identified 58 problems: 52 were due to model inconsistencies (discrepancies between requirements and design), 4 were safety issues detected via HCSP simulations (such as trains failing to stop as expected), and 2 stemmed from miscommunications leading to misunderstandings of requirements.

Therefore, to streamline our efforts and rigorously prove the consistency of this safety-critical system, we explore various methods for model synchronization and verification. Table 5 outlines the costs before synchronization (Construction Costs), the expenses for maintaining and verifying consistency (Usage Costs), and the benefits resulting from synchronization and verification. Among these approaches, KBX offers an avenue for achieving verified synchronization with reduced costs while maintaining the trustworthiness of the formal method.

Using KBX, we first construct the HCSP to UML $\mathbb{K}$ transformation. Then, the synthesizer generates the UML and HCSP BX $\mathbb{K}$ transformation definitions, as described in Sections 4.2 and 4.3. These definitions are used for synchronization and consistency verification, as per Section 4.4.

Table 6 illustrates the features we have supported for HCSP and plantUML. The first row denotes the features supported for HCSP, while the second row pertains to the features supported for plantUML. While some features may not be included, the supported set consists of all crucial HCSP features, sufficient for modeling high-speed

[13]Various alternative implementations of the proof check are in https://us.metamath.org/other.html#verifiers; also includes 350 lines of Python, 400 lines of Haskell, etc., accessed on July 1st, 2024

[14]https://github.com/seL4/isabelle/blob/master/src/Pure/thm.ML

[15]https://github.com/coq/coq/tree/master/checker

[16]https://github.com/leanprover/lean4/tree/master/src/kernel

Table 5. Comparison of Consistency Maintenance and Verification Approaches

Approach	Construction Costs	Usage Costs	Benefits
Manual	(1) Consistency definition documentation.	(1) Manual model synchronization. (2) Consistency documentation.	Traceable consistency of manual safeguards
Verified Translator	(1) translator from HCSP to UML. (2) translator from UML to HCSP. (3) verification of the translators.	(1) Automatic translation	Verified consistency without complements
Forward simulation & BX	(1) BX program for HCSP and UML synchronization.	(1) Automatic synchronization. (2) Verification and (3) Correction of the synchronized models.	Verified consistency
Formal BX	(1) formal specifications of forward transformation and (2) backward transformation (3) verification of round-tripping laws	(1) Automatic synchronization with formal validation.	Verified consistency
KBX	(1) formal unidirectional transformation specification	(1) Automatic synchronization with formal validation.	Verified consistency

Table 6. Support statistics for HCSP and plantUML.

HCSP	skip	assignment	channel	wait	conditional	interrupt	hybrid
plantUML	message	comment	loop	option	alt	group	color

maglev train partition handover behavior. Next, we outline some of the technical steps involved in constructing the HCSP and UML BX: (1) Developing a unidirectional transformation from HCSP to plantUML. (2) Synthesizing a bidirectional transformation definition between HCSP and plantUML. (3) Performing synchronization and verification using these definitions.

Developing a unidirectional transformation from HCSP to plantUML. First, we construct a transformation definition from HCSP to PlantUML within the $\mathbb{K}$ framework. We opt for this approach because it simplifies the process by starting with deterministic semantics while also offering ample support for HCSP features. This definition encompasses the syntax of both HCSP and PlantUML, defines the program configuration to describe the state structure and initial state, and formulates rewrite rules to specify the relation between HCSP and PlantUML.

Program configuration. Fig. 13 visually represents the configuration related to M, N, and S discussed in Section 4.1. The HCSP programs to transform are contained in *hcsp*, while the transformed models in UML are stored in *uml*. The state S is instantiated by *tmp* and *threads*, initialized with empty list and map.

$$\big\langle \langle hcsp\rangle_{hcsp} \;\; \langle uml\rangle_{uml} \;\; \langle list\rangle_{tmp} \;\; \langle id \mapsto k\rangle_{threads} \big\rangle_{bx}$$

Fig. 13. Transformation Configuration for UML and HCSP BX.

Rewrite rules. Constructing these rules presents challenges owing to semantic disparities. For instance, both HCSP and UML sequence diagrams incorporate communication, but HCSP applies channels for message transmission, while UML dispatches messages directly to objects. This divergence becomes even more intricate when addressing loops, interrupts, and other functionalities. Therefore, we introduce some of HCSP's concurrent semantics (i.e. spawn and join) to accurately describe the transformation process. The following phases offer a glimpse into our semantics.

(1) *Spawn*: The first phase spawns all the processes from *hcsp* to *threads*. Variables are assigned as follows: *id* is set to the process identifier, *k* is set to the statements of the process.
(2) *Transformation*: The second phase constructs the transformation from HCSP statements to UML "statements", which is the key part of the transformation specification.
(3) *Join*: The third phase join processes into *tmp*, and delete the corresponding threads in *threads*.
(4) *Termination*: In the final phase, the synchronized list in *tmp* is moved to *uml* and converted into a PlantUML sequence diagram.

Table 7. Size of the transformation definitions.

	Unidirectional	Forward	Backward	Total Generated
N words	1139	2019	2053	4072

Synthesizing a bidirectional transformation between HCSP and plantUML. Second, KBX derives bidirectional transformation definitions from the given unidirectional transformation definition. Table 7 provides a comparison between transformations produced manually and those generated by KBX. By handling complements and round-tripping laws—typically the most challenging aspects—KBX simplifies the development process and alleviates users' burdens. Notably, KBX reduces code volume by 72%, calculated as follows:

$$\frac{TotalGenerated - Unidirectional}{TotalGenerated}$$

Performing synchronization and verification using these definitions. Finally, we employ the $\mathbb{K}$ framework to generate tools from these generated BX definitions for synchronization and verification. These tools are executable and verifiable, facilitating synchronization while generating proof hints during the process. Existing works [Chen et al. 2021a; Lin et al. 2023] take these hints and the $\mathbb{K}$ definitions as inputs to produce proofs. Thus, we can achieve automatic model synchronization and simultaneous verification of their consistency.

Table 8. Performance of KBX.

Process	Time (s)
BX Definitions Generation	0.02
Tool Generation	24.18
Forward Creation (10 LOC)	5.20
Forward Creation (100 LOC)	2.21
Forward Creation (1000 LOC)	6.26
Forward Creation (Proof Hints)	8.96
Forward Synchronization	6.82
Backward Synchronization	4.58

Experimental Setup: Apple M2, 16GB RAM, macOS Sonoma 14.5, $\mathbb{K}$ 7.1.23.

Table 8 presents the performance of KBX in generating BX definitions and using the $\mathbb{K}$ framework to generate tools for synchronization and verification. The results show that KBX is efficient in generating BX definitions.

The tool generation process, which uses the "kompile" command in $\mathbb{K}$, takes more time but is acceptable since it only needs to be performed once. Forward creation measures the time taken to convert HCSP to PlantUML without complements, showing times of 5.2, 2.2, and 6.2 seconds for 10, 100, and 1000 lines of HCSP, respectively. Although synchronization times indeed correlate with the complexity of the user-defined $\mathbb{K}$ definitions and the scale of the synchronization targets, the longer time for 10 lines compared to 100 lines, and the similar times for 100 and 1000 lines, suggest that parsing and printing, rather than rewriting, might be the most time-consuming phases. These times are expected to improve with ongoing optimizations in $\mathbb{K}$. Nonetheless, our primary goal is not real-time synchronization but the generation of proofs for consistency verification in safety-critical systems. In this context, the time required to generate proof hints and perform forward and backward synchronization with complements is acceptable.

In summary, **KBX is effective in the development of real-world formal HCSP and plantUML BX, significantly improving the efficiency of this process.**

5.4 Threats to Validity

Here, we discuss several threats that may affect the validity of our evaluation, following the typical classifications: construct validity, conclusion validity, internal validity, and external validity [Wohlin et al. 2012].

Construct Validity. The primary goal of this paper is verified synchronization for safety-critical systems, focusing on both the expressiveness and trustworthiness of synchronization and verification. Thus, the threat to construct validity is whether our RQs and metrics fully and accurately cover our objectives. To mitigate this, we ensure the first two RQs focus on synchronization and verification, while the third RQ demonstrates practical applicability. To ensure the detailed evaluation metrics align with our primary goal, we reviewed the synchronization and verification workflows to obtain a comprehensive set of metrics, and checked the accuracy of these metrics through rational analysis. Specifically, we checked the following rationale for different types of metrics: (1) ease of development and review for synchronization expressiveness; (2) application of formal methods and formal verification for synchronization trustworthiness; (3) consistency definitions as a correctness specification for verification expressiveness; and (4) the trust base size for verification trustworthiness.

Conclusion Validity. Threats to conclusion validity concern the reliability of the methods used to derive results and infer conclusions. To obtain reliable results for metrics, we used peer-reviewed literature where possible and provided replication packages and official links to ensure reproducibility. For accurate conclusions, we evaluated qualitative metrics for RQ1, supplemented by quantitative metrics, because quantitative analysis is not directly aligned with our primary goal. For instance, code length does not equate to expressiveness, and performance does not impact trustworthiness. In RQ2, we analyzed KBX's workflow, theory, and implementation, comparing the trust base sizes. Conclusions for RQ3 were based on statistical comparisons between generated and human-written definitions.

Internal Validity. Internal validity threats pertain to dependencies between evaluation factors and KBX. RQ1 remains unaffected as our method is theoretically based on matching logic and implemented using $\mathbb{K}$, which are distinct from other BX frameworks. For RQ2, the expressiveness and trustworthiness of $\mathbb{K}$ could threaten implementation validity. However, our theory's foundation in matching logic facilitates future migration to other theorem provers, because matching logic can be constructed within other theorem provers like Lean [17]. RQ3 presents no internal validity threats since both our theory and implementation are language-independent.

External Validity. Threats to external validity concern the applicability of our method in broader scenarios. To mitigate these threats, we employed the third-party Families2Persons benchmark and provided a complex HCSP and PlantUML. Additionally, the formal framework $\mathbb{K}$ on which our method is based has been used for complex languages like C and Java, supporting its applicability to diverse languages.

[17] https://gitlab.com/ilds/aml-lean/MatchingLogic/, accessed on July 1, 2024

To minimize selection bias, we adopted several measures. In RQ1, we included BX frameworks from [Anjorin et al. 2020] and extended two works focusing on expressiveness. We did not extend works on trustworthiness because no BX frameworks using theorem provers other than BiGUL were found. In RQ2, we compared widely used theorem provers—Coq, Lean, and Isabelle/HOL—noting that other theorem provers should not differ significantly in trust base.

6 RELATED WORK

Consistency Verification. To verify complete systems, the utilization of multiple models is imperative due to limitations imposed by the language, the intricacy of the system itself, and the need to adhere to certain standards [Kulik et al. 2022; Reid et al. 2020]. Researchers have employed various approaches, such as refinement verification, translation validation, and verified translators, to guarantee the consistency of these models.

Refinement verification [Back and Wright 2012; Morgan et al. 1988] is a common mechanism for ensuring consistency through forward simulation. For instance, [Chen et al. 2018; Klein et al. 2010; Li et al. 2021] verify the consistency of models across various abstract levels, ensuring the correctness of the operating system, device driver, and hypervisor. Translation validation is another approach for consistency verification [Bang et al. 2022; Lopes et al. 2021; Sewell et al. 2013], which automatically validates the correctness of the compilation process, ensuring the consistency between compilation sources and targets. Previous studies [Lano et al. 2015; Leinenbach 2008; Leroy 2012; Srivastava et al. 2010] proposed transformation verification methods to obtain verified translators. These approaches can automatically convert models while rigorously ensuring consistency. However, they only offer unidirectional transformations without the capability to recover missing information or perform reverse transformations.

Bidirectional Transformation. In contrast to these formal verification methods, BX frameworks typically focus on implementing model synchronization to maintain consistency. Consistency in BX refers to whether a pair of state-based transformation operations or delta-based propagation operations conform to specific criteria (e.g., well-behavedness as defined in [Hidaka et al. 2016]). This differs from formal verification, which generally ensures that two models comply with a specific relation within their respective semantics.

Regarding expressiveness, BiYacc [Zhu et al. 2015] presents a language-oriented language to roll the parser and reflective printer into one. Hobit [Matsuda and Wang 2018] enhances expressiveness by eliminating lens combinators, providing greater flexibility in representation. Hinkel and Burger [2019] (i.e., NMF) took a different approach by reusing the C# language to design a domain-specific language for bidirectional programming. We adopt the $\mathbb{K}$ definition to allow users to define formal consistency intuitively, without needing lens combinators or implementing parsers and printers.

In terms of trustworthiness, BX frameworks often prioritize rational synchronization over typical verification or validation, which aims to ensure that transformations conform to their specifications and accurately reflect user requirements. For example, BiGUL [Ko et al. 2016] employs formal verification in BX to ensure the recovery of missing information, i.e., round-trip or well-behavedness. As noted by [Anjorin et al. 2020; Hildebrandt et al. 2013], tools like [Hermann et al. 2011] and eMoflon [Weidmann et al. 2019] define algorithmic correctness based on TGG (Triple Graph Grammars) semantics and provide proofs, although not within theorem provers. However, TGG is merely an abstract formalism, and users need a specific language to construct one. For example, BXtend [Buchmann 2018] offers an internal DSL based on Xtend, which lacks formal semantics like those of K-Java [Bogdanas and Roşu 2015]. Furthermore, algorithmic correctness does not guarantee implementation or execution correctness. Tools like JTL [Cicchetti et al. 2011] and EVL+Strace [Samimi-Dehkordi et al. 2018] define consistency through constraints using ASP (Answer Set Programming) and EVL (Epsilon Validation Language), enabling synchronization result checks. However, their trust base is significantly larger than that proposed by theorem proving. For example, ASP solvers introduce a substantial trust base (over 100,000 lines of code[18]). Moreover, BX programs not constructed within a formal framework cannot be formally verified. In contrast, KBX aims to embed all aspects of BX within a formal framework, enabling the construction of complex, verifiable formal consistency definitions and providing consistency verification through proof generation with a small trust base.

To enhance the efficiency of developing BX programs, researchers have dedicated efforts to synthesizing BX through various approaches. For instance, Yamaguchi et al. [2021] use unidirectional sketches to synthesize bidirectional programs, while Maina et al. [2018]; Miltner et al. [2018, 2019] employ a pair of regular expressions and example input-output pairs to achieve this automation. Given a partial specification, these algorithms are designed to narrow down the search space for identifying a suitable BX program within the context of existing BX frameworks. Compared to these recent works, KBX differs in that (1) it supports more flexible syntaxes and focuses on synthesizing formal BX specifications (recall that the actual transformers are generated by $\mathbb{K}$); (2) it uses formal unidirectional transformation specifications, without the need for additional input-output examples; (3) it rests on our backward rewriting for deterministic and efficient synthesis, instead of previous type-directed enumerative algorithms.

7 DISCUSSION

Compared to previous BX frameworks, KBX deeply integrates formal methods and BX towards a minimal trust base for verified synchronization in safety-critical systems. The proof generation ensures synchronization conforms to formal consistency definitions, offering consistency verification comparable to widely-used refinement verification in high-assurance evaluations. However, our approach has some limitations that future work could address.

First, the trust base can be further reduced. Currently, our prototype requires users to trust either our implementation or the generated bidirectional transformation definition along with the proof checker. Ideally, users should only need to trust the consistency definition and proof checker. Future work could involve implementing verified KBX within formal frameworks like K or Lean.

Second, our work does not yet consider the usability and performance features present in other BX frameworks, such as incremental transformation and conflict resolution. Incorporating these features could enhance the performance and usability of KBX. Based on the feature-based classification proposed by [Hidaka et al. 2016], KBX can be categorized as follows: (1) *Technical Space*: Textual Artifacts; (2) *Correspondence*: Allow missing information on both sides, user-defined coverage; (3) *Definition Directionality*: Users provide unidirectional definitions, and KBX generates bidirectional transformation definitions; (4) *Expressiveness*: Turing complete; (5) *Changes*: State-based representation, offline input, complete support; (6) *Execution*: Enforcement semantics, automatic execution, full application, semantic approach, explicit backward transformation, explicit traces, trace out of models, standard traces, complement out of models; (7) *Well-Behavedness*: acceptability, consistency, composability, preservation, propagation, undoability, functional behavior.

Third, verifying informal models, such as images and natural language, poses a significant challenge for theorem-proving-based verification methods, including KBX. For instance, we ensure consistency between HCSP programs and PlantUML programs but do not verify the consistency between PlantUML and the final rendered image. Although this approach suffices for our motivating example, an ideal solution would involve representing informal models as verifiable mathematical models instead of opaque vectors or matrices. Progress in AI explainability and formalization research may gradually address this issue.

Fourth, our implementation does not fully support all $\mathbb{K}$ language features, which requires further enhancement. For example, the *hook* attribute binds syntax productions to specific C++ implementations or SMTLib in $\mathbb{K}$. This affects the automatic synthesis of bidirectional transformation definitions for certain $\mathbb{K}$ built-ins, like "+Int".

[18]https://github.com/potassco/clingo, accessed on July 1, 2024. JTL uses DLV https://dlv.demacs.unical.it/home, which is not open source.

In such cases, users need to manually adjust the generated definitions according to our theory. Specifically, synchronizing " X +Int Y " with its result " ?Z " is akin to synchronizing "log(A); L := R ;" with " P -[C]-> P : A ". Both lose information on both sides; however, the latter's right-hand side is a symbolic variable " ?Z ". After modifying the generated definitions according to Sections 4.2 and 4.3, the forward transformation of "1 +Int 1" should yield "2" with "$ruleid \mapsto (1, 1)$" stored in the complements holder. The backward transformation should use these complements to recover "1 +Int 1". When no complements are available, it should return a default value like "0 +Int 0" or a symbolic value like $\{Z|Z == 2\}$.

This highlights another potential improvement area: program synthesis. The default value "0 +Int 0" should be avoided as it only equals 0, suggesting the need for symbolic defaults. However, concrete values may be necessary at times. In such cases, an SMT solver can generate a suitable value, verified through theorem proving. Without deep integration with formal frameworks, current BX frameworks cannot achieve this. Even advanced bidirectional evaluation approaches like [Lubin et al. 2020; Mayer et al. 2018; Zhang et al. 2023; Zhang and Hu 2022] do not consider program synthesis. They typically propose a BX between HTML and their custom languages, placing placeholders (akin to KBX's default value) in incomplete parts. Compared to KBX's case study, these custom languages lack concurrency semantics, so their dynamic semantics complexity is lower than HCSP; and HTML is similar to PlantUML, differing in rendering results. This demonstrates the difficulty and importance of incorporating expressiveness for semantics, even informal ones, into bidirectional transformations. For example, [Zhang et al. 2023] provided informal static semantics for the object-oriented feature in bidirectional transformation in 2023, whereas complete formal semantics of Java [Bogdanas and Roşu 2015] and JavaScript [Park et al. 2015] were defined in $\mathbb{K}$ as early as 2015. These semantics can be reused in KBX for defining bidirectional transformations of object-oriented languages.

8 CONCLUSION

We have introduced KBX, the first formal bidirectional transformation framework for verified model synchronization in diverse languages and abstraction levels. Our evaluation shows that KBX outperforms other BX frameworks while maintaining the verification ability of the $\mathbb{K}$ framework with consistency verification outcomes comparable to refinement verification. Furthermore, we demonstrate the usability of KBX using a real-world application of HCSP & UML BX. For future work, we plan to apply KBX to more expressive languages and complicated specifications, thereby expanding its versatility and addressing more challenging scenarios.

REFERENCES

María Alpuente, Daniel Pardo, and Alicia Villanueva. 2020. Abstract Contract Synthesis and Verification in the Symbolic $\mathbb{K}$ Framework. *Fundamenta Informaticae* 177, 3-4 (Dec. 2020), 235–273. https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/FI-2020-1989

Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. 2020. Benchmarking Bidirectional Transformations: Theory, Implementation, Application, and Assessment. *Software and systems modeling* 19, 3 (2020), 647–691.

Ralph-Johan Back and Joakim Wright. 2012. *Refinement Calculus: A Systematic Introduction.* Springer Science & Business Media.

Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. SMT-Based Translation Validation for Machine Learning Compiler. In *Computer Aided Verification.* Springer, Cham, 386–407. https://link.springer.com/chapter/10.1007/978-3-031-13188-2_19

Ron Bell. 2006. Introduction to IEC 61508. In *Acm International Conference Proceeding Series*, Vol. 162. 3–12.

Lorenzo Bettini and Sven Efftinge. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend: Learn How to Implement a DSL with Xtext and Xtend Using Easy-to-Understand Examples and Best Practices* (second edition ed.). Packt Publishing, Birmingham Mumbai.

Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, Mumbai India, 445–456. https://dl.acm.org/doi/10.1145/2676726.2676982

Thomas Buchmann. 2018. BXtend-A Framework for (Bidirectional) Incremental Model Transformations.. In *MODELSWARD.* 336–345.

Thomas Buchmann, Matthias Bank, and Bernhard Westfechtel. 2022. BXtendDSL: A Layered Framework for Bidirectional Model Transformations Combining a Declarative and an Imperative Language. *Journal of Systems and Software* 189 (July 2022), 111288. https://www.sciencedirect.com/science/article/pii/S0164121222000462

Zhou Chaochen, Wang Ji, and Anders P. Ravn. 1996. A Formal Description of Hybrid Systems. In *Hybrid Systems III: Verification and Control 3*. Springer, 511–530.

Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2018. Toward Compositional Verification of Interruptible Os Kernels and Device Drivers. *Journal of Automated Reasoning* 61, 1 (2018), 141–189.

Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. 2021a. Towards a Trustworthy Semantics-Based Language Framework via Proof Generation. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 477–499.

Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. 2021b. Matching Logic Explained. *Journal of Logical and Algebraic Methods in Programming* 120 (April 2021), 100638. https://linkinghub.elsevier.com/retrieve/pii/S2352220821000018

Xiaohong Chen and Grigore Roşu. 2019. Matching μ-Logic. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–13.

Xiaohong Chen and Grigore Rosu. 2019. Matching Mu-Logic: Foundation of K Framework. In *8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2011. JTL: A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering (Lecture Notes in Computer Science)*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer, Berlin, Heidelberg, 183–202.

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A Complete Formal Semantics of X86-64 User-Level Instruction Set Architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Phoenix AZ USA, 1133–1148. https://dl.acm.org/doi/10.1145/3314221.3314601

Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. *ACM SIGPLAN Notices* 47, 1 (2012), 533–544.

Chucky M. Ellison and Grigore Roşu. 2011. An Executable Formal Semantics of C with Applications: Technical Report. *Acm Sigplan Notices* (2011). http://www.researchgate.net/publication/49175991_A_Formal_Semantics_of_C_with_Applications_Technical_Report

Romina Eramo, Alfonso Pierantonio, and Michele Tucci. 2018. Enhancing the JTL Tool for Bidirectional Transformations. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. ACM, Nice France, 36–41. https://dl.acm.org/doi/10.1145/3191697.3191720

Chris Hathhorn, Chucky Ellison, and Grigore Ro?u. 2015. Defining the Undefinedness of C. *ACM* (2015), 336–345. http://dl.acm.org/doi/abs/10.1145/2737924.2737979

Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. 2011. Correctness of Model Synchronization Based on Triple Graph Grammars. In *Model Driven Engineering Languages and Systems*, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Vol. 6981. Springer Berlin Heidelberg, Berlin, Heidelberg, 668–682. http://link.springer.com/10.1007/978-3-642-24485-8_49

Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. 2016. Feature-Based Classification of Bidirectional Transformation Approaches. *Software & Systems Modeling* 15, 3 (July 2016), 907–928. http://link.springer.com/10.1007/s10270-014-0450-0

Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. 2013. A Survey of Triple Graph Grammar Tools. *Electronic Communications of the EASST* 57 (2013). https://eceasst.org/index.php/eceasst/article/view/2083

Georg Hinkel and Erik Burger. 2019. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Software & Systems Modeling* 18, 1 (2019), 249–278.

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric Lenses. *ACM SIGPLAN Notices* 46, 1 (2011), 371–384.

Zhenjiang Hu and Hsiang-Shang Ko. 2018. Principles and Practice of Bidirectional Programming in BiGUL. In *Bidirectional Transformations*, Jeremy Gibbons and Perdita Stevens (Eds.). Vol. 9715. Springer International Publishing, Cham, 100–150. http://link.springer.com/10.1007/978-3-319-79108-1_4

Stephen Jacklin. 2012. Certification of Safety-Critical Software Under DO-178C and DO-278A. In *Infotech@Aerospace 2012*. American Institute of Aeronautics and Astronautics, Garden Grove, California. https://arc.aiaa.org/doi/10.2514/6.2012-2473

Daisuke Kinoshita and Keisuke Nakano. 2017. Bidirectional Certified Programming.. In *BX@ ETAPS*. 31–38.

Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling Consistency in View-Based System Development—the Vitruvius Approach. *Journal of Systems and Software* 171 (2021), 110815.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. (2014). http://trustworthy.systems/publications/nictaabstracts/Klein_AEMSKH_14.abstract, /publications/nictaabstracts/Klein_AEMSKH_14.abstract

Gerwin Klein, Thomas Sewell, and Simon Winwood. 2010. Refinement in the Formal Verification of the seL4 Microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 323–339.

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, St. Petersburg FL USA, 61–72. https://dl.acm.org/doi/10.1145/2847538.2847544

Dexter Kozen. 1983. Results on the Propositional μ-Calculus. *Theoretical computer science* 27, 3 (1983), 333–354.

Tomas Kulik, Brijesh Dongol, Peter Gorm Larsen, Hugo Daniel Macedo, Steve Schneider, Peter WV Tran-Jørgensen, and James Woodcock. 2022. A Survey of Practical Formal Methods for Security. *Formal Aspects of Computing* 34, 1 (2022), 1–39.

Kevin Lano, Tony Clark, and S. Kolahdouz-Rahimi. 2015. A Framework for Model Transformation Verification. *Formal Aspects of Computing* 27 (2015), 193–235.

Dirk Carsten Leinenbach. 2008. Compiler Verification in the Context of Pervasive System Verification. (2008).

Xavier Leroy. 2012. Mechanized Semantics for Compiler Verification. In *Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings 10*. Springer, 386–388.

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1782–1799. https://ieeexplore.ieee.org/document/9519433/

Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 56–84. https://dl.acm.org/doi/10.1145/3586029

Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. 2010. A Calculus for Hybrid CSP. In *Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28-December 1, 2010. Proceedings 8*. Springer, 1–15.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 65–79. https://dl.acm.org/doi/10.1145/3453483.3454030

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–29. https://dl.acm.org/doi/10.1145/3408991

Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing Quotient Lenses. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 1–29. https://dl.acm.org/doi/10.1145/3236775

Kazutaka Matsuda and Meng Wang. 2018. Hobit: Programming Lenses without Using Lens Combinators. In *European Symposium on Programming*. Springer, 31–59.

Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–28. https://dl.acm.org/doi/10.1145/3276497

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing Bijective Lenses. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–30. https://dl.acm.org/doi/10.1145/3158089

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing Symmetric Lenses. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 1–28. https://dl.acm.org/doi/10.1145/3341699

Carroll Morgan, Ken Robinson, and Paul Gardiner. 1988. *On the Refinement Calculus*. Number 70 in Technical Monograph / Oxford Univ. Computing Laboratory, Programming Research Group. University Computing Laboratory, Oxford.

Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. 2020. Towards Making Formal Methods Normal: Meeting Developers Where They Are. *arXiv preprint arXiv:2010.16345* (2020). arXiv:2010.16345

Leila Samimi-Dehkordi, Bahman Zamani, and Shekoufeh Kolahdouz-Rahimi. 2018. EVL+Strace: A Novel Bidirectional Model Transformation Approach. *Information and Software Technology* 100 (Aug. 2018), 47–72. https://linkinghub.elsevier.com/retrieve/pii/S0950584917300629

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 471–482.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 313–326.

Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, Amsterdam Netherlands, 74–91. https://dl.acm.org/doi/10.1145/2983990.2984027

Perdita Stevens. 2008. A Landscape of Bidirectional Model Transformations. In *Generative and Transformational Techniques in Software Engineering II*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Vol. 5235. Springer Berlin Heidelberg, Berlin, Heidelberg, 408–424. http://link.springer.com/10.1007/978-3-540-88643-3_10

Nils Weidmann, Anthony Anjorin, Gergely Varro, Lars Fritsche, Andy Schurr, and Erhan Leblebici. 2019. Incremental Bidirectional Model Transformation with eMoflon::IBeX. In *Proceedings of the Eighth International Workshop on Bidirectional Transformations*. http://ceur-ws.org

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Vol. 236. Springer.

Xiong Xu, Shuling Wang, Bohua Zhan, Xiangyu Jin, Jean-Pierre Talpin, and Naijun Zhan. 2022. Unified Graphical Co-Modeling, Analysis and Verification of Cyber-Physical Systems by Combining AADL and Simulink/Stateflow. *Theoretical computer science* 903 (2022), 1–25.

Masaomi Yamaguchi, Kazutaka Matsuda, Cristina David, and Meng Wang. 2021. Synbit: Synthesizing Bidirectional Programs Using Unidirectional Sketches. arXiv:2108.13783 [cs] http://arxiv.org/abs/2108.13783

Bohua Zhan, Bin Gu, Xiong Xu, Xiangyu Jin, Shuling Wang, Bai Xue, Xiaofeng Li, Yao Chen, Mengfei Yang, and Naijun Zhan. 2021. Brief Industry Paper: Modeling and Verification of Descent Guidance Control of Mars Lander. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 457–460.

Xing Zhang, Guanchen Guo, Xiao He, and Zhenjiang Hu. 2023. Bidirectional Object-Oriented Programming: Towards Programmatic and Direct Manipulation of Objects. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 230–255. https://dl.acm.org/doi/10.1145/3586035

Xing Zhang and Zhenjiang Hu. 2022. Towards Bidirectional Live Programming for Incomplete Programs. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2154–2164. https://dl.acm.org/doi/10.1145/3510003.3510195

Zirun Zhu, Hsiang-Shang Ko, Pedro Miguel Ribeiro Martins, João Alexandre Saraiva, and Zhenjiang Hu. 2015. BiYacc: Roll Your Parser and Reflective Printer into One. In *Proceedings of the Fourth International Workshop on Bidirectional Transformations*. CEUR-Ws. http://ceur-ws.org

Zirun Zhu, Hsiang-Shang Ko, Yongzhe Zhang, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2020. Unifying Parsing and Reflective Printing for Fully Disambiguated Grammars. *New Generation Computing* 38, 3 (July 2020), 423–476. https://doi.org/10.1007/s00354-019-00082-y

Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. 2015. Formal Verification of Simulink/Stateflow Diagrams. In *Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings 13*. Springer, 464–481.

Liang Zou, Naijun Zhany, Shuling Wang, Martin Fränzle, and Shengchao Qin. 2013. Verifying Simulink Diagrams via a Hybrid Hoare Logic Prover. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, 1–10.